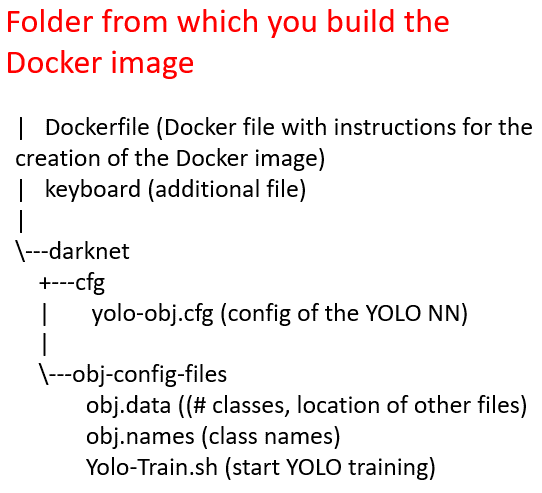
# Prepare the Images and the annotations for YOLO

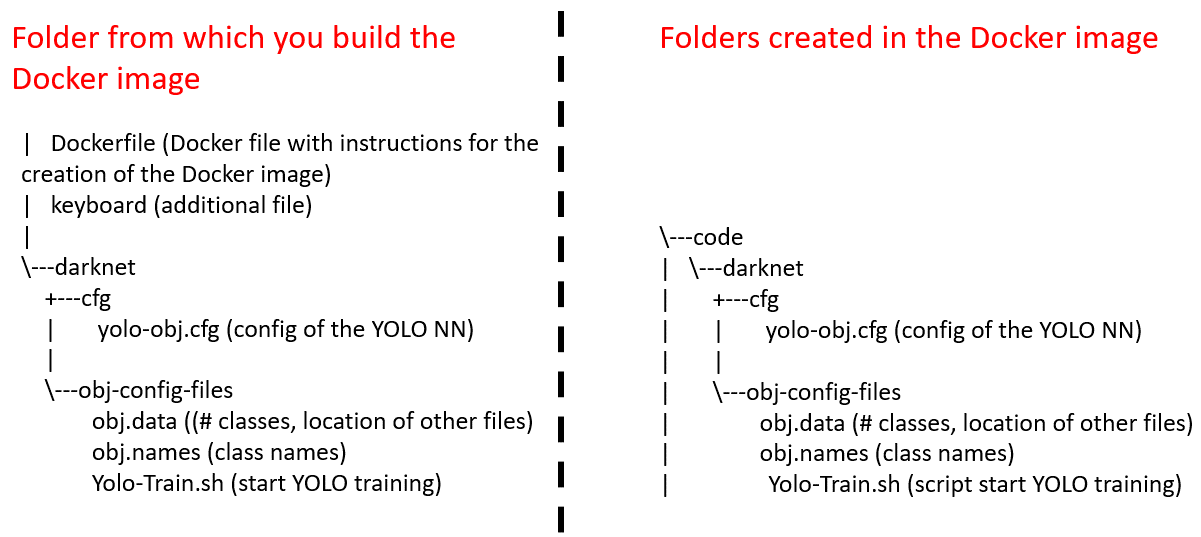
1. Annotate images with VGG Annotator - http://www.robots.ox.ac.uk/~vgg/software/via/
2. Save the annotations in csv file
   1. See example Docker-Yolo/VGGvsYolo-Annotations/VGGAnnotation-example.csv
3. Run the script VGG-2-Yolo-annotations.ipynb to generate the:
   1. <imageId>.txt file with YOLO Compatible annotations for <imageId>.png
      1. See for example
         1. Docker-Yolo/VGGvsYolo-Annotations/outPut-VGG-2-Yolo-annotations/ 1998.png
         2. Docker-Yolo/VGGvsYolo-Annotations/outPut-VGG-2-Yolo-annotations/ 1998.txt
   2. Generate a txt file that list the path (path within the Docker container) of for the images
      1. Docker-Yolo/VGGvsYolo-Annotations/outPut-VGG-2-Yolo-annotations/ 1998.txt
      2. Note: You will have to split this text file to generate the train.txt and test.txt files, each of them with the list of images (and their path on the Docker image) for train and test respectively
4. The above files will be later on uploaded on the EC2 where the docker image runs – see section 3

# Creation of the Docker image for YOLOV4

1. Deploy an EC2 with the following setup:
   * EC2 Type: P2
   * AMI: Deep Learning Base AMI (Ubuntu 18.04) Version 30.0
   * IAM Role: Grant full access to S3 Buckets
   * Public IP: make sure to be able to SSH into the machine
2. Install Docker
   * E.g. “quick-dirty” solution
   * >> curl -fsSL https://get.docker.com -o get-docker.sh
   * >> sudo sh get-docker.sh
3. Create on the EC2 a folder:
   * Save in this folder the files and folders of Docker-Yolo/creation-yolo-image-P2 on this github
   * The content of the folder must be like below



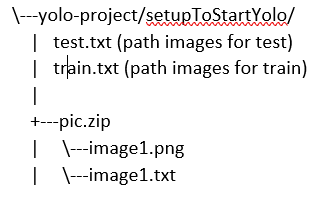
* + Please check the content of Dockerfile to get acquainted about what operations will be performed by Docker when building the image
  + From this folder run (note the dot at the end of the command)
    - >> docker build -t <container-name> .
  + For example (note the dot at the end of the command)
    - >> docker build -t yolo-container .
  + The final result will be:



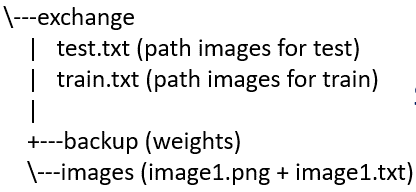
1. Optionally you can tag your container and push it to your Docker repository
   * >> docker tag yolo-container <docker-account-name>/<custom-image-name>
   * >> docker login
     + Type your user and then the pwd
   * >> docker push <docker-account-name>/<custom-image-name>

# Run the Docker Image with Yolo

1. Create an S3 bucket named //yolo-project/
   * Create a folder //yolo-project/setupToStartYolo/
   * Upload on the S3 the images and annotations produced after section 1:
     + A zip file that has
       - The images
       - For each image, the corresponding txt file with YOLO compatible annotations
     + The train.txt and test.txt files with the paths to the images
   * Eg. The structure on the S3 is going to be



1. Deploy an EC2 with the following setup:
   * EC2 Type: P2
   * AMI: Deep Learning Base AMI (Ubuntu 18.04) Version 30.0
   * IAM Role: Grant full access to S3 Buckets 🡺 this is needed to transfer files
   * Public IP: make sure to be able to SSH into the machine
   * Execute the User Data Script Docker-Yolo/ec2-User-Script-for-Yolo.sh
     + Note: we are assuming that a docker image with YOLO named is available in one Docker repository – the script refers specifically to the image asonnellini/yolo-custom-folders, you should change it to point it to the image of your interest
   * Once the script is executed:
     + You will finally have the following folders/files on the EC2

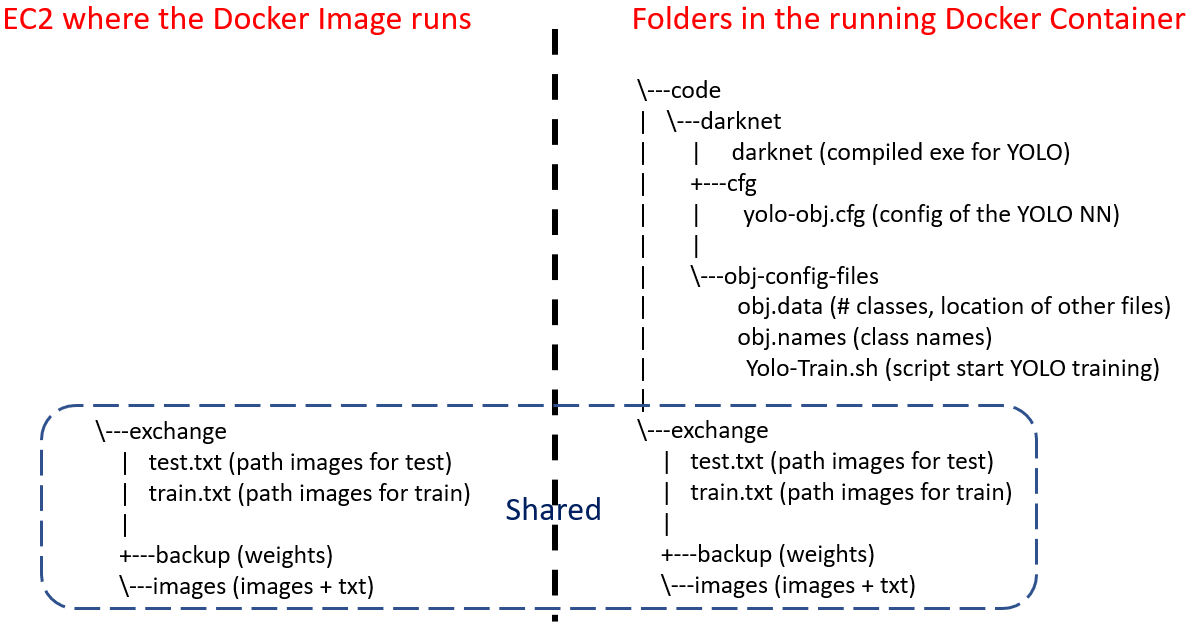


* + - The docker image with YOLO will be available on the EC2
  + Run the Docker image executing the following command – see the next point for the detached mode:
    - >> sudo docker run -it -p 80:8090 --gpus all -v ~/exchange:/exchange asonnellini/yolo-custom-folders
      * Given that the container was run with -i and -t, you can detach from it and leave it running using the CTRL-p CTRL-q key sequence
        + To re-attach/re-enter the container:

>> sudo docker attach ddc081f03827

>> sudo docker container ps

* + - * --gpus all : ensures the Docker Image can use all the gpus and that the folder ~/exchange on the EC2 is shared with the folder /exchange on the Docker Image
      * -p 80:8090 maps the container port 8090 to the host 80
  + At this point in time:
    - You will be prompted inside the Docker Image
    - The setup shown below is in place



* + - To ensure everything is fine you can try to run a test:
      * Download some weights for the coco.dataset:
        + >> wget https://github.com/AlexeyAB/darknet/releases/download/darknet\_yolo\_v3\_optimal/yolov4.weights
      * Run a test detection
        + >>>> ./darknet detector test ./cfg/coco.data ./cfg/yolov4.cfg ./yolov4.weights data/dog.jpg -thresh 0.25

# Train YOLO

From “inside” the Docker Image:

* Adjust the parameters in yolo-obj.cfg file to match your needs (e.g. batch size, height and width)
* Start a run executing for example the script /code/darknet/obj-config-files/Yolo-Train.sh:
  + >> /code/darknet/darknet detector train /code/darknet/obj-config-files/obj.data /code/darknet/cfg/yolo-obj.cfg /code/darknet/yolov4.conv.137 -dont\_show -mjpeg\_port 8090 -map
* During the training YOLO will dump in the folder /exchange/backup the weights every 100 iterations
* You can check the GPU memory consumption running:
  + >> nvidia-smi

# Predict with YOLO

From “inside” the Docker image:

* Change in /code/darknet/cfg/yolo-obj.cfg:
  + Uncomment the part about testing (#batch=1 #subdivisions=1)
  + Comment the part about training (batch=64 subdivisions=16)
* Run the below command – the output of the detection will be dumped in /exchange/result.txt
  + >> /code/darknet/darknet detector test /code/darknet/obj-config-files/obj.data /code/darknet/cfg/yolo-obj.cfg /exchange/backup/yolo-obj\_last.weights /exchange/images/9732\_AnteroPosterior\_unspecified.png -thresh 0.25 -ext\_output > /exchange/result.txt

# Integrate YOLO with a FLASK API and trigger the detection via POST

To integrate flask in the docker image created for yolo, we created a new docker image asonnellini/yolo-custom-folders**-flask.**

The Docker image asonnellini/yolo-custom-folders**-flask** is identical to asonnellini/yolo-custom-folders but includes a flask API.

To use it:

* Download the docker image asonnellini/yolo-custom-folders-flask
  + >> docker pull asonnellini/yolo-custom-folders-flask
* Run the docker container with the following command:
  + >> sudo docker run -d --rm -p 8090:8090 --gpus all -v ~/exchange:/exchange yolo-custom-folders-flask python3 darknet/flask-API/flask\_api.py
  + The above command
    - runs the container in detached mode
    - maps port 8090 on the EC2 host to container port 8090 – please check out this article https://pythonspeed.com/articles/docker-connection-refused/ for some additional info about what’s needed to successfully run the flask API on docker
    - runs the script flask\_api.py (inside the container) which is the script that has the flask API
* The flask API implemented in the container
  + Listens to all the container network interfaces and port 8090
  + exposes 1 endpoint / and 2 methods, namely GET and POST:
    - POST + / = triggers the detection on an image saved in the S3 bucket
    - GET + / = returns just some info about the flask API
  + Specifically, to run a detection on an image, the following has to be setup:
    - Ensure the EC2 where the docker container is running has:
      * an IAM that grants full access to S3 buckets
      * a Security Group that allows traffic on the port 8090
    - An S3 bucket that saves the image on which the detection has to be run
    - Another S3 bucket available to store the image upon detection
    - Run a curl command toward the flask API endpoint / passing information in a json, e.g.:
      * >> curl -H "Content-Type: application/json" -X POST -d '{"bucketName": "yolo-project", "folderBucket": "toDetect", "imgFileName": "1998\_AnteroPosterior\_supine.png", "bucketDestination": "yolo-project", "bucketDestFolder": "detected"}' http://<private IP of EC2>:8090/

Where for the json all the following mandatory attributes must be specified:

* + - * bucketName: name of the S3 bucket that hosts the image on which detection has to be run
      * folderBucker: folder in the S3 bucket where the file is placed; if the file is not in any folder this attribute must be set to empty string “”
      * imgFileName: name of the image saved on the S3 bucket
      * bucketDestination: name of the bucket which will host the post-detection image
      * bucketDestFolder: name of the folder in the bucket which will host the post-detection image; if the file is not in any folder this attribute must be set to empty string “”
  + if the detection is executed successfully, the flask API will reply to the POST message with details about the path where the post-detection image is stored on an S3 :

{ "Outcome": "OK",

"destBucket": "yolo-project",

"destBucketFolder": "detected",

"destFileName": "123-det\_1998\_AnteroPosterior\_supine.png"

}

Note: at this stage the detection is not performed for real, the flask API currently mimics just the mechanism of getting an image from an S3 bucket and copy another image on another S3 bucket as per the information passed via the POST command.

The new image asonnellini/yolo-custom-folders-flask was created starting from the same dockerfile used for image asonnellini/yolo-custom-folders,adding to it the following:

* python modules:
  + flask\_RESTfull
  + boto3
* Inside the folder /code/darknet a folder named flasked\_API that has the files
  + flask\_api.py – has the flask API code
  + uploadDownload.py - has some functions used by the flask API code
  + darknet.py and darknet\_images.py – has some functions to call the detection

All the above files can be found in the folder yolo-and-flask on this github page.

Points yet to be implemented:

* implement the detection (you will have to modify flask API and save in the docker image the final weights)
* implement a mechanism that allows to have unique id for each image
  + to consider the case for example where we have
    - multiple docker containers running detection and all of them saving images on the same S3 bicket in the same folders
      * it would be necessary to mark each image from each container with an Id that is made of
        + unique Id of the docker container
        + unique Id produced by one container for its own images