# YOLOv5 by Ultralytics, AGPL-3.0 license

"""

Train a YOLOv5 model on a custom dataset.

Models and datasets download automatically from the latest YOLOv5 release.

Usage - Single-GPU training:

$ python train.py --data coco128.yaml --weights yolov5s.pt --img 640 # from pretrained (recommended)

$ python train.py --data coco128.yaml --weights '' --cfg yolov5s.yaml --img 640 # from scratch

Usage - Multi-GPU DDP training:

$ python -m torch.distributed.run --nproc\_per\_node 4 --master\_port 1 train.py --data coco128.yaml --weights yolov5s.pt --img 640 --device 0,1,2,3

Models: https://github.com/ultralytics/yolov5/tree/master/models

Datasets: https://github.com/ultralytics/yolov5/tree/master/data

Tutorial: https://docs.ultralytics.com/yolov5/tutorials/train\_custom\_data

"""

import argparse

import math

import os

import random

import subprocess

import sys

import time

from copy import deepcopy

from datetime import datetime

from pathlib import Path

try:

import comet\_ml # must be imported before torch (if installed)

except ImportError:

comet\_ml = None

import numpy as np

import torch

import torch.distributed as dist

import torch.nn as nn

import yaml

from torch.optim import lr\_scheduler

from tqdm import tqdm

FILE = Path(\_\_file\_\_).resolve()

ROOT = FILE.parents[0] # YOLOv5 root directory

if str(ROOT) not in sys.path:

sys.path.append(str(ROOT)) # add ROOT to PATH

ROOT = Path(os.path.relpath(ROOT, Path.cwd())) # relative

import val as validate # for end-of-epoch mAP

from models.experimental import attempt\_load

from models.yolo import Model

from utils.autoanchor import check\_anchors

from utils.autobatch import check\_train\_batch\_size

from utils.callbacks import Callbacks

from utils.dataloaders import create\_dataloader

from utils.downloads import attempt\_download, is\_url

from utils.general import (LOGGER, TQDM\_BAR\_FORMAT, check\_amp, check\_dataset, check\_file, check\_git\_info,

check\_git\_status, check\_img\_size, check\_requirements, check\_suffix, check\_yaml, colorstr,

get\_latest\_run, increment\_path, init\_seeds, intersect\_dicts, labels\_to\_class\_weights,

labels\_to\_image\_weights, methods, one\_cycle, print\_args, print\_mutation, strip\_optimizer,

yaml\_save)

from utils.loggers import Loggers

from utils.loggers.comet.comet\_utils import check\_comet\_resume

from utils.loss import ComputeLoss

from utils.metrics import fitness

from utils.plots import plot\_evolve

from utils.torch\_utils import (EarlyStopping, ModelEMA, de\_parallel, select\_device, smart\_DDP, smart\_optimizer,

smart\_resume, torch\_distributed\_zero\_first)

LOCAL\_RANK = int(os.getenv('LOCAL\_RANK', -1)) # https://pytorch.org/docs/stable/elastic/run.html

RANK = int(os.getenv('RANK', -1))

WORLD\_SIZE = int(os.getenv('WORLD\_SIZE', 1))

GIT\_INFO = check\_git\_info()

def train(hyp, opt, device, callbacks): # hyp is path/to/hyp.yaml or hyp dictionary

save\_dir, epochs, batch\_size, weights, single\_cls, evolve, data, cfg, resume, noval, nosave, workers, freeze = \

Path(opt.save\_dir), opt.epochs, opt.batch\_size, opt.weights, opt.single\_cls, opt.evolve, opt.data, opt.cfg, \

opt.resume, opt.noval, opt.nosave, opt.workers, opt.freeze

callbacks.run('on\_pretrain\_routine\_start')

# Directories

w = save\_dir / 'weights' # weights dir

(w.parent if evolve else w).mkdir(parents=True, exist\_ok=True) # make dir

last, best = w / 'last.pt', w / 'best.pt'

# Hyperparameters

if isinstance(hyp, str):

with open(hyp, errors='ignore') as f:

hyp = yaml.safe\_load(f) # load hyps dict

LOGGER.info(colorstr('hyperparameters: ') + ', '.join(f'{k}={v}' for k, v in hyp.items()))

opt.hyp = hyp.copy() # for saving hyps to checkpoints

# Save run settings

if not evolve:

yaml\_save(save\_dir / 'hyp.yaml', hyp)

yaml\_save(save\_dir / 'opt.yaml', vars(opt))

# Loggers

data\_dict = None

if RANK in {-1, 0}:

loggers = Loggers(save\_dir, weights, opt, hyp, LOGGER) # loggers instance

# Register actions

for k in methods(loggers):

callbacks.register\_action(k, callback=getattr(loggers, k))

# Process custom dataset artifact link

data\_dict = loggers.remote\_dataset

if resume: # If resuming runs from remote artifact

weights, epochs, hyp, batch\_size = opt.weights, opt.epochs, opt.hyp, opt.batch\_size

# Config

plots = not evolve and not opt.noplots # create plots

cuda = device.type != 'cpu'

init\_seeds(opt.seed + 1 + RANK, deterministic=True)

with torch\_distributed\_zero\_first(LOCAL\_RANK):

data\_dict = data\_dict or check\_dataset(data) # check if None

train\_path, val\_path = data\_dict['train'], data\_dict['val']

nc = 1 if single\_cls else int(data\_dict['nc']) # number of classes

names = {0: 'item'} if single\_cls and len(data\_dict['names']) != 1 else data\_dict['names'] # class names

is\_coco = isinstance(val\_path, str) and val\_path.endswith('coco/val2017.txt') # COCO dataset

# Model

check\_suffix(weights, '.pt') # check weights

pretrained = weights.endswith('.pt')

if pretrained:

with torch\_distributed\_zero\_first(LOCAL\_RANK):

weights = attempt\_download(weights) # download if not found locally

ckpt = torch.load(weights, map\_location='cpu') # load checkpoint to CPU to avoid CUDA memory leak

model = Model(cfg or ckpt['model'].yaml, ch=3, nc=nc, anchors=hyp.get('anchors')).to(device) # create

exclude = ['anchor'] if (cfg or hyp.get('anchors')) and not resume else [] # exclude keys

csd = ckpt['model'].float().state\_dict() # checkpoint state\_dict as FP32

csd = intersect\_dicts(csd, model.state\_dict(), exclude=exclude) # intersect

model.load\_state\_dict(csd, strict=False) # load

LOGGER.info(f'Transferred {len(csd)}/{len(model.state\_dict())} items from {weights}') # report

else:

model = Model(cfg, ch=3, nc=nc, anchors=hyp.get('anchors')).to(device) # create

amp = check\_amp(model) # check AMP

# Freeze

freeze = [f'model.{x}.' for x in (freeze if len(freeze) > 1 else range(freeze[0]))] # layers to freeze

for k, v in model.named\_parameters():

v.requires\_grad = True # train all layers

# v.register\_hook(lambda x: torch.nan\_to\_num(x)) # NaN to 0 (commented for erratic training results)

if any(x in k for x in freeze):

LOGGER.info(f'freezing {k}')

v.requires\_grad = False

# Image size

gs = max(int(model.stride.max()), 32) # grid size (max stride)

imgsz = check\_img\_size(opt.imgsz, gs, floor=gs \* 2) # verify imgsz is gs-multiple

# Batch size

if RANK == -1 and batch\_size == -1: # single-GPU only, estimate best batch size

batch\_size = check\_train\_batch\_size(model, imgsz, amp)

loggers.on\_params\_update({'batch\_size': batch\_size})

# Optimizer

nbs = 64 # nominal batch size

accumulate = max(round(nbs / batch\_size), 1) # accumulate loss before optimizing

hyp['weight\_decay'] \*= batch\_size \* accumulate / nbs # scale weight\_decay

optimizer = smart\_optimizer(model, opt.optimizer, hyp['lr0'], hyp['momentum'], hyp['weight\_decay'])

# Scheduler

if opt.cos\_lr:

lf = one\_cycle(1, hyp['lrf'], epochs) # cosine 1->hyp['lrf']

else:

lf = lambda x: (1 - x / epochs) \* (1.0 - hyp['lrf']) + hyp['lrf'] # linear

scheduler = lr\_scheduler.LambdaLR(optimizer, lr\_lambda=lf) # plot\_lr\_scheduler(optimizer, scheduler, epochs)

# EMA

ema = ModelEMA(model) if RANK in {-1, 0} else None

# Resume

best\_fitness, start\_epoch = 0.0, 0

if pretrained:

if resume:

best\_fitness, start\_epoch, epochs = smart\_resume(ckpt, optimizer, ema, weights, epochs, resume)

del ckpt, csd

# DP mode

if cuda and RANK == -1 and torch.cuda.device\_count() > 1:

LOGGER.warning(

'WARNING ⚠️ DP not recommended, use torch.distributed.run for best DDP Multi-GPU results.\n'

'See Multi-GPU Tutorial at https://docs.ultralytics.com/yolov5/tutorials/multi\_gpu\_training to get started.'

)

model = torch.nn.DataParallel(model)

# SyncBatchNorm

if opt.sync\_bn and cuda and RANK != -1:

model = torch.nn.SyncBatchNorm.convert\_sync\_batchnorm(model).to(device)

LOGGER.info('Using SyncBatchNorm()')

# Trainloader

train\_loader, dataset = create\_dataloader(train\_path,

imgsz,

batch\_size // WORLD\_SIZE,

gs,

single\_cls,

hyp=hyp,

augment=True,

cache=None if opt.cache == 'val' else opt.cache,

rect=opt.rect,

rank=LOCAL\_RANK,

workers=workers,

image\_weights=opt.image\_weights,

quad=opt.quad,

prefix=colorstr('train: '),

shuffle=True,

seed=opt.seed)

labels = np.concatenate(dataset.labels, 0)

mlc = int(labels[:, 0].max()) # max label class

assert mlc < nc, f'Label class {mlc} exceeds nc={nc} in {data}. Possible class labels are 0-{nc - 1}'

# Process 0

if RANK in {-1, 0}:

val\_loader = create\_dataloader(val\_path,

imgsz,

batch\_size // WORLD\_SIZE \* 2,

gs,

single\_cls,

hyp=hyp,

cache=None if noval else opt.cache,

rect=True,

rank=-1,

workers=workers \* 2,

pad=0.5,

prefix=colorstr('val: '))[0]

if not resume:

if not opt.noautoanchor:

check\_anchors(dataset, model=model, thr=hyp['anchor\_t'], imgsz=imgsz) # run AutoAnchor

model.half().float() # pre-reduce anchor precision

callbacks.run('on\_pretrain\_routine\_end', labels, names)

# DDP mode

if cuda and RANK != -1:

model = smart\_DDP(model)

# Model attributes

nl = de\_parallel(model).model[-1].nl # number of detection layers (to scale hyps)

hyp['box'] \*= 3 / nl # scale to layers

hyp['cls'] \*= nc / 80 \* 3 / nl # scale to classes and layers

hyp['obj'] \*= (imgsz / 640) \*\* 2 \* 3 / nl # scale to image size and layers

hyp['label\_smoothing'] = opt.label\_smoothing

model.nc = nc # attach number of classes to model

model.hyp = hyp # attach hyperparameters to model

model.class\_weights = labels\_to\_class\_weights(dataset.labels, nc).to(device) \* nc # attach class weights

model.names = names

# Start training

t0 = time.time()

nb = len(train\_loader) # number of batches

nw = max(round(hyp['warmup\_epochs'] \* nb), 100) # number of warmup iterations, max(3 epochs, 100 iterations)

# nw = min(nw, (epochs - start\_epoch) / 2 \* nb) # limit warmup to < 1/2 of training

last\_opt\_step = -1

maps = np.zeros(nc) # mAP per class

results = (0, 0, 0, 0, 0, 0, 0) # P, R, mAP@.5, mAP@.5-.95, val\_loss(box, obj, cls)

scheduler.last\_epoch = start\_epoch - 1 # do not move

scaler = torch.cuda.amp.GradScaler(enabled=amp)

stopper, stop = EarlyStopping(patience=opt.patience), False

compute\_loss = ComputeLoss(model) # init loss class

callbacks.run('on\_train\_start')

LOGGER.info(f'Image sizes {imgsz} train, {imgsz} val\n'

f'Using {train\_loader.num\_workers \* WORLD\_SIZE} dataloader workers\n'

f"Logging results to {colorstr('bold', save\_dir)}\n"

f'Starting training for {epochs} epochs...')

for epoch in range(start\_epoch, epochs): # epoch ------------------------------------------------------------------

callbacks.run('on\_train\_epoch\_start')

model.train()

# Update image weights (optional, single-GPU only)

if opt.image\_weights:

cw = model.class\_weights.cpu().numpy() \* (1 - maps) \*\* 2 / nc # class weights

iw = labels\_to\_image\_weights(dataset.labels, nc=nc, class\_weights=cw) # image weights

dataset.indices = random.choices(range(dataset.n), weights=iw, k=dataset.n) # rand weighted idx

# Update mosaic border (optional)

# b = int(random.uniform(0.25 \* imgsz, 0.75 \* imgsz + gs) // gs \* gs)

# dataset.mosaic\_border = [b - imgsz, -b] # height, width borders

mloss = torch.zeros(3, device=device) # mean losses

if RANK != -1:

train\_loader.sampler.set\_epoch(epoch)

pbar = enumerate(train\_loader)

LOGGER.info(('\n' + '%11s' \* 7) % ('Epoch', 'GPU\_mem', 'box\_loss', 'obj\_loss', 'cls\_loss', 'Instances', 'Size'))

if RANK in {-1, 0}:

pbar = tqdm(pbar, total=nb, bar\_format=TQDM\_BAR\_FORMAT) # progress bar

optimizer.zero\_grad()

for i, (imgs, targets, paths, \_) in pbar: # batch -------------------------------------------------------------

callbacks.run('on\_train\_batch\_start')

ni = i + nb \* epoch # number integrated batches (since train start)

imgs = imgs.to(device, non\_blocking=True).float() / 255 # uint8 to float32, 0-255 to 0.0-1.0

# Warmup

if ni <= nw:

xi = [0, nw] # x interp

# compute\_loss.gr = np.interp(ni, xi, [0.0, 1.0]) # iou loss ratio (obj\_loss = 1.0 or iou)

accumulate = max(1, np.interp(ni, xi, [1, nbs / batch\_size]).round())

for j, x in enumerate(optimizer.param\_groups):

# bias lr falls from 0.1 to lr0, all other lrs rise from 0.0 to lr0

x['lr'] = np.interp(ni, xi, [hyp['warmup\_bias\_lr'] if j == 0 else 0.0, x['initial\_lr'] \* lf(epoch)])

if 'momentum' in x:

x['momentum'] = np.interp(ni, xi, [hyp['warmup\_momentum'], hyp['momentum']])

# Multi-scale

if opt.multi\_scale:

sz = random.randrange(int(imgsz \* 0.5), int(imgsz \* 1.5) + gs) // gs \* gs # size

sf = sz / max(imgs.shape[2:]) # scale factor

if sf != 1:

ns = [math.ceil(x \* sf / gs) \* gs for x in imgs.shape[2:]] # new shape (stretched to gs-multiple)

imgs = nn.functional.interpolate(imgs, size=ns, mode='bilinear', align\_corners=False)

# Forward

with torch.cuda.amp.autocast(amp):

pred = model(imgs) # forward

loss, loss\_items = compute\_loss(pred, targets.to(device)) # loss scaled by batch\_size

if RANK != -1:

loss \*= WORLD\_SIZE # gradient averaged between devices in DDP mode

if opt.quad:

loss \*= 4.

# Backward

scaler.scale(loss).backward()

# Optimize - https://pytorch.org/docs/master/notes/amp\_examples.html

if ni - last\_opt\_step >= accumulate:

scaler.unscale\_(optimizer) # unscale gradients

torch.nn.utils.clip\_grad\_norm\_(model.parameters(), max\_norm=10.0) # clip gradients

scaler.step(optimizer) # optimizer.step

scaler.update()

optimizer.zero\_grad()

if ema:

ema.update(model)

last\_opt\_step = ni

# Log

if RANK in {-1, 0}:

mloss = (mloss \* i + loss\_items) / (i + 1) # update mean losses

mem = f'{torch.cuda.memory\_reserved() / 1E9 if torch.cuda.is\_available() else 0:.3g}G' # (GB)

pbar.set\_description(('%11s' \* 2 + '%11.4g' \* 5) %

(f'{epoch}/{epochs - 1}', mem, \*mloss, targets.shape[0], imgs.shape[-1]))

callbacks.run('on\_train\_batch\_end', model, ni, imgs, targets, paths, list(mloss))

if callbacks.stop\_training:

return

# end batch ------------------------------------------------------------------------------------------------

# Scheduler

lr = [x['lr'] for x in optimizer.param\_groups] # for loggers

scheduler.step()

if RANK in {-1, 0}:

# mAP

callbacks.run('on\_train\_epoch\_end', epoch=epoch)

ema.update\_attr(model, include=['yaml', 'nc', 'hyp', 'names', 'stride', 'class\_weights'])

final\_epoch = (epoch + 1 == epochs) or stopper.possible\_stop

if not noval or final\_epoch: # Calculate mAP

results, maps, \_ = validate.run(data\_dict,

batch\_size=batch\_size // WORLD\_SIZE \* 2,

imgsz=imgsz,

half=amp,

model=ema.ema,

single\_cls=single\_cls,

dataloader=val\_loader,

save\_dir=save\_dir,

plots=False,

callbacks=callbacks,

compute\_loss=compute\_loss)

# Update best mAP

fi = fitness(np.array(results).reshape(1, -1)) # weighted combination of [P, R, mAP@.5, mAP@.5-.95]

stop = stopper(epoch=epoch, fitness=fi) # early stop check

if fi > best\_fitness:

best\_fitness = fi

log\_vals = list(mloss) + list(results) + lr

callbacks.run('on\_fit\_epoch\_end', log\_vals, epoch, best\_fitness, fi)

# Save model

if (not nosave) or (final\_epoch and not evolve): # if save

ckpt = {

'epoch': epoch,

'best\_fitness': best\_fitness,

'model': deepcopy(de\_parallel(model)).half(),

'ema': deepcopy(ema.ema).half(),

'updates': ema.updates,

'optimizer': optimizer.state\_dict(),

'opt': vars(opt),

'git': GIT\_INFO, # {remote, branch, commit} if a git repo

'date': datetime.now().isoformat()}

# Save last, best and delete

torch.save(ckpt, last)

if best\_fitness == fi:

torch.save(ckpt, best)

if opt.save\_period > 0 and epoch % opt.save\_period == 0:

torch.save(ckpt, w / f'epoch{epoch}.pt')

del ckpt

callbacks.run('on\_model\_save', last, epoch, final\_epoch, best\_fitness, fi)

# EarlyStopping

if RANK != -1: # if DDP training

broadcast\_list = [stop if RANK == 0 else None]

dist.broadcast\_object\_list(broadcast\_list, 0) # broadcast 'stop' to all ranks

if RANK != 0:

stop = broadcast\_list[0]

if stop:

break # must break all DDP ranks

# end epoch ----------------------------------------------------------------------------------------------------

# end training -----------------------------------------------------------------------------------------------------

if RANK in {-1, 0}:

LOGGER.info(f'\n{epoch - start\_epoch + 1} epochs completed in {(time.time() - t0) / 3600:.3f} hours.')

for f in last, best:

if f.exists():

strip\_optimizer(f) # strip optimizers

if f is best:

LOGGER.info(f'\nValidating {f}...')

results, \_, \_ = validate.run(

data\_dict,

batch\_size=batch\_size // WORLD\_SIZE \* 2,

imgsz=imgsz,

model=attempt\_load(f, device).half(),

iou\_thres=0.65 if is\_coco else 0.60, # best pycocotools at iou 0.65

single\_cls=single\_cls,

dataloader=val\_loader,

save\_dir=save\_dir,

save\_json=is\_coco,

verbose=True,

plots=plots,

callbacks=callbacks,

compute\_loss=compute\_loss) # val best model with plots

if is\_coco:

callbacks.run('on\_fit\_epoch\_end', list(mloss) + list(results) + lr, epoch, best\_fitness, fi)

callbacks.run('on\_train\_end', last, best, epoch, results)

torch.cuda.empty\_cache()

return results

def parse\_opt(known=False):

parser = argparse.ArgumentParser()

parser.add\_argument('--weights', type=str, default=ROOT/'yolov5s.pt', help='initial weights path')

parser.add\_argument('--cfg', type=str, default=ROOT/'wzry/wzry\_model.yaml', help='model.yaml path')

parser.add\_argument('--data', type=str, default=ROOT/'wzry/wzry\_parameter.yaml', help='dataset.yaml path')

parser.add\_argument('--hyp', type=str, default=ROOT/'data/hyps/hyp.scratch-low.yaml', help='hyperparameters path')

parser.add\_argument('--epochs', type=int, default=200, help='total training epochs')

parser.add\_argument('--batch-size', type=int, default=4, help='total batch size for all GPUs, -1 for autobatch')

parser.add\_argument('--imgsz', '--img', '--img-size', type=int, default=192, help='train, val image size (pixels)')

parser.add\_argument('--rect', action='store\_true', help='rectangular training')

parser.add\_argument('--resume', nargs='?', const=True, default=False, help='resume most recent training')

parser.add\_argument('--nosave', action='store\_true', help='only save final checkpoint')

parser.add\_argument('--noval', action='store\_true', help='only validate final epoch')

parser.add\_argument('--noautoanchor', action='store\_true', help='disable AutoAnchor')

parser.add\_argument('--noplots', action='store\_true', help='save no plot files')

parser.add\_argument('--evolve', type=int, nargs='?', const=300, help='evolve hyperparameters for x generations')

parser.add\_argument('--bucket', type=str, default='', help='gsutil bucket')

parser.add\_argument('--cache', type=str, nargs='?', const='ram', help='image --cache ram/disk')

parser.add\_argument('--image-weights', action='store\_true', help='use weighted image selection for training')

parser.add\_argument('--device', default='cpu', help='cuda device, i.e. 0 or 0,1,2,3 or cpu')

parser.add\_argument('--multi-scale', action='store\_true', help='vary img-size +/- 50%%')

parser.add\_argument('--single-cls', action='store\_true', help='train multi-class data as single-class')

parser.add\_argument('--optimizer', type=str, choices=['SGD', 'Adam', 'AdamW'], default='SGD', help='optimizer')

parser.add\_argument('--sync-bn', action='store\_true', help='use SyncBatchNorm, only available in DDP mode')

parser.add\_argument('--workers', type=int, default=4, help='max dataloader workers (per RANK in DDP mode)')

parser.add\_argument('--project', default=ROOT / 'runs/train', help='save to project/name')

parser.add\_argument('--name', default='exp', help='save to project/name')

parser.add\_argument('--exist-ok', action='store\_true', help='existing project/name ok, do not increment')

parser.add\_argument('--quad', action='store\_true', help='quad dataloader')

parser.add\_argument('--cos-lr', action='store\_true', help='cosine LR scheduler')

parser.add\_argument('--label-smoothing', type=float, default=0.0, help='Label smoothing epsilon')

parser.add\_argument('--patience', type=int, default=100, help='EarlyStopping patience (epochs without improvement)')

parser.add\_argument('--freeze', nargs='+', type=int, default=[0], help='Freeze layers: backbone=10, first3=0 1 2')

parser.add\_argument('--save-period', type=int, default=-1, help='Save checkpoint every x epochs (disabled if < 1)')

parser.add\_argument('--seed', type=int, default=0, help='Global training seed')

parser.add\_argument('--local\_rank', type=int, default=-1, help='Automatic DDP Multi-GPU argument, do not modify')

# Logger arguments

parser.add\_argument('--entity', default=None, help='Entity')

parser.add\_argument('--upload\_dataset', nargs='?', const=True, default=False, help='Upload data, "val" option')

parser.add\_argument('--bbox\_interval', type=int, default=-1, help='Set bounding-box image logging interval')

parser.add\_argument('--artifact\_alias', type=str, default='latest', help='Version of dataset artifact to use')

return parser.parse\_known\_args()[0] if known else parser.parse\_args()

def main(opt, callbacks=Callbacks()):

# Checks

if RANK in {-1, 0}:

print\_args(vars(opt))

check\_git\_status()

check\_requirements(ROOT / 'requirements.txt')

# Resume (from specified or most recent last.pt)

if opt.resume and not check\_comet\_resume(opt) and not opt.evolve:

last = Path(check\_file(opt.resume) if isinstance(opt.resume, str) else get\_latest\_run())

opt\_yaml = last.parent.parent / 'opt.yaml' # train options yaml

opt\_data = opt.data # original dataset

if opt\_yaml.is\_file():

with open(opt\_yaml, errors='ignore') as f:

d = yaml.safe\_load(f)

else:

d = torch.load(last, map\_location='cpu')['opt']

opt = argparse.Namespace(\*\*d) # replace

opt.cfg, opt.weights, opt.resume = '', str(last), True # reinstate

if is\_url(opt\_data):

opt.data = check\_file(opt\_data) # avoid HUB resume auth timeout

else:

opt.data, opt.cfg, opt.hyp, opt.weights, opt.project = \

check\_file(opt.data), check\_yaml(opt.cfg), check\_yaml(opt.hyp), str(opt.weights), str(opt.project) # checks

assert len(opt.cfg) or len(opt.weights), 'either --cfg or --weights must be specified'

if opt.evolve:

if opt.project == str(ROOT / 'runs/train'): # if default project name, rename to runs/evolve

opt.project = str(ROOT / 'runs/evolve')

opt.exist\_ok, opt.resume = opt.resume, False # pass resume to exist\_ok and disable resume

if opt.name == 'cfg':

opt.name = Path(opt.cfg).stem # use model.yaml as name

opt.save\_dir = str(increment\_path(Path(opt.project) / opt.name, exist\_ok=opt.exist\_ok))

# DDP mode

device = select\_device(opt.device, batch\_size=opt.batch\_size)

if LOCAL\_RANK != -1:

msg = 'is not compatible with YOLOv5 Multi-GPU DDP training'

assert not opt.image\_weights, f'--image-weights {msg}'

assert not opt.evolve, f'--evolve {msg}'

assert opt.batch\_size != -1, f'AutoBatch with --batch-size -1 {msg}, please pass a valid --batch-size'

assert opt.batch\_size % WORLD\_SIZE == 0, f'--batch-size {opt.batch\_size} must be multiple of WORLD\_SIZE'

assert torch.cuda.device\_count() > LOCAL\_RANK, 'insufficient CUDA devices for DDP command'

torch.cuda.set\_device(LOCAL\_RANK)

device = torch.device('cuda', LOCAL\_RANK)

dist.init\_process\_group(backend='nccl' if dist.is\_nccl\_available() else 'gloo')

# Train

if not opt.evolve:

train(opt.hyp, opt, device, callbacks)

# Evolve hyperparameters (optional)

else:

# Hyperparameter evolution metadata (mutation scale 0-1, lower\_limit, upper\_limit)

meta = {

'lr0': (1, 1e-5, 1e-1), # initial learning rate (SGD=1E-2, Adam=1E-3)

'lrf': (1, 0.01, 1.0), # final OneCycleLR learning rate (lr0 \* lrf)

'momentum': (0.3, 0.6, 0.98), # SGD momentum/Adam beta1

'weight\_decay': (1, 0.0, 0.001), # optimizer weight decay

'warmup\_epochs': (1, 0.0, 5.0), # warmup epochs (fractions ok)

'warmup\_momentum': (1, 0.0, 0.95), # warmup initial momentum

'warmup\_bias\_lr': (1, 0.0, 0.2), # warmup initial bias lr

'box': (1, 0.02, 0.2), # box loss gain

'cls': (1, 0.2, 4.0), # cls loss gain

'cls\_pw': (1, 0.5, 2.0), # cls BCELoss positive\_weight

'obj': (1, 0.2, 4.0), # obj loss gain (scale with pixels)

'obj\_pw': (1, 0.5, 2.0), # obj BCELoss positive\_weight

'iou\_t': (0, 0.1, 0.7), # IoU training threshold

'anchor\_t': (1, 2.0, 8.0), # anchor-multiple threshold

'anchors': (2, 2.0, 10.0), # anchors per output grid (0 to ignore)

'fl\_gamma': (0, 0.0, 2.0), # focal loss gamma (efficientDet default gamma=1.5)

'hsv\_h': (1, 0.0, 0.1), # image HSV-Hue augmentation (fraction)

'hsv\_s': (1, 0.0, 0.9), # image HSV-Saturation augmentation (fraction)

'hsv\_v': (1, 0.0, 0.9), # image HSV-Value augmentation (fraction)

'degrees': (1, 0.0, 45.0), # image rotation (+/- deg)

'translate': (1, 0.0, 0.9), # image translation (+/- fraction)

'scale': (1, 0.0, 0.9), # image scale (+/- gain)

'shear': (1, 0.0, 10.0), # image shear (+/- deg)

'perspective': (0, 0.0, 0.001), # image perspective (+/- fraction), range 0-0.001

'flipud': (1, 0.0, 1.0), # image flip up-down (probability)

'fliplr': (0, 0.0, 1.0), # image flip left-right (probability)

'mosaic': (1, 0.0, 1.0), # image mixup (probability)

'mixup': (1, 0.0, 1.0), # image mixup (probability)

'copy\_paste': (1, 0.0, 1.0)} # segment copy-paste (probability)

with open(opt.hyp, errors='ignore') as f:

hyp = yaml.safe\_load(f) # load hyps dict

if 'anchors' not in hyp: # anchors commented in hyp.yaml

hyp['anchors'] = 3

if opt.noautoanchor:

del hyp['anchors'], meta['anchors']

opt.noval, opt.nosave, save\_dir = True, True, Path(opt.save\_dir) # only val/save final epoch

# ei = [isinstance(x, (int, float)) for x in hyp.values()] # evolvable indices

evolve\_yaml, evolve\_csv = save\_dir / 'hyp\_evolve.yaml', save\_dir / 'evolve.csv'

if opt.bucket:

# download evolve.csv if exists

subprocess.run([

'gsutil',

'cp',

f'gs://{opt.bucket}/evolve.csv',

str(evolve\_csv), ])

for \_ in range(opt.evolve): # generations to evolve

if evolve\_csv.exists(): # if evolve.csv exists: select best hyps and mutate

# Select parent(s)

parent = 'single' # parent selection method: 'single' or 'weighted'

x = np.loadtxt(evolve\_csv, ndmin=2, delimiter=',', skiprows=1)

n = min(5, len(x)) # number of previous results to consider

x = x[np.argsort(-fitness(x))][:n] # top n mutations

w = fitness(x) - fitness(x).min() + 1E-6 # weights (sum > 0)

if parent == 'single' or len(x) == 1:

# x = x[random.randint(0, n - 1)] # random selection

x = x[random.choices(range(n), weights=w)[0]] # weighted selection

elif parent == 'weighted':

x = (x \* w.reshape(n, 1)).sum(0) / w.sum() # weighted combination

# Mutate

mp, s = 0.8, 0.2 # mutation probability, sigma

npr = np.random

npr.seed(int(time.time()))

g = np.array([meta[k][0] for k in hyp.keys()]) # gains 0-1

ng = len(meta)

v = np.ones(ng)

while all(v == 1): # mutate until a change occurs (prevent duplicates)

v = (g \* (npr.random(ng) < mp) \* npr.randn(ng) \* npr.random() \* s + 1).clip(0.3, 3.0)

for i, k in enumerate(hyp.keys()): # plt.hist(v.ravel(), 300)

hyp[k] = float(x[i + 7] \* v[i]) # mutate

# Constrain to limits

for k, v in meta.items():

hyp[k] = max(hyp[k], v[1]) # lower limit

hyp[k] = min(hyp[k], v[2]) # upper limit

hyp[k] = round(hyp[k], 5) # significant digits

# Train mutation

results = train(hyp.copy(), opt, device, callbacks)

callbacks = Callbacks()

# Write mutation results

keys = ('metrics/precision', 'metrics/recall', 'metrics/mAP\_0.5', 'metrics/mAP\_0.5:0.95', 'val/box\_loss',

'val/obj\_loss', 'val/cls\_loss')

print\_mutation(keys, results, hyp.copy(), save\_dir, opt.bucket)

# Plot results

plot\_evolve(evolve\_csv)

LOGGER.info(f'Hyperparameter evolution finished {opt.evolve} generations\n'

f"Results saved to {colorstr('bold', save\_dir)}\n"

f'Usage example: $ python train.py --hyp {evolve\_yaml}')

def run(\*\*kwargs):

# Usage: import train; train.run(data='coco128.yaml', imgsz=320, weights='yolov5m.pt')

opt = parse\_opt(True)

for k, v in kwargs.items():

setattr(opt, k, v)

main(opt)

return opt

if \_\_name\_\_ == '\_\_main\_\_':

opt = parse\_opt()

main(opt)