**ПРИЛОЖЕНИЕ А**

УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ

«БРЕСТСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

# КАФЕДРА ИНТЕЛЛЕКТУАЛЬНЫХ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

ДЕТЕКЦИЯ И ИДЕНТИФИКАЦИЯ ЛЮДЕЙ НА ФОТО- И ВИДЕОИЗОБРАЖЕНИЯХ

**ТЕКСТ ПРОГРАММЫ**

КР.ПО4.190335-04 12 00

## Листов 13

|  |  |
| --- | --- |
| Руководитель | П. А. Кочурко |
| Выполнил | А. В. Грицак |
| Консультант |  |
| по ЕСПД | П. А. Кочурко |
|  |  |

2021

**СОДЕРЖАНИЕ**

1. train.py – файл для обучения нейронной сети с использованием библиотеки PyTorch.
2. detect.py – файл для распознавания тела и лица человека на основе обученной модели с использованием библиотеки PyTorch.
3. coco.yaml – файл для формирования описания модели нейронной сети.
4. coco.names – файл для формирования описание названий классов.
5. train.py

import argparse  
import math  
import os  
import random  
import sys  
import time  
from copy import deepcopy  
from datetime import datetime  
from pathlib import Path  
import numpy as np  
import torch  
import torch.distributed as dist  
import torch.nn as nn  
import yaml  
from torch.cuda import amp  
from torch.nn.parallel import DistributedDataParallel as DDP  
from torch.optim import SGD, Adam, 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 # 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.datasets import create\_dataloader  
from utils.downloads import attempt\_download  
from utils.general import (LOGGER, check\_dataset, check\_file, 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)  
from utils.loggers import Loggers  
from utils.loggers.wandb.wandb\_utils import check\_wandb\_resume  
from utils.loss import ComputeLoss  
from utils.metrics import fitness  
from utils.plots import plot\_evolve, plot\_labels  
from utils.torch\_utils import EarlyStopping, ModelEMA, de\_parallel, select\_device, torch\_distributed\_zero\_first  
  
LOCAL\_RANK = int(os.getenv('LOCAL\_RANK', -1))  
RANK = int(os.getenv('RANK', -1))  
WORLD\_SIZE = int(os.getenv('WORLD\_SIZE', 1))  
  
def train(hyp, # path/to/hyp.yaml or hyp dictionary  
opt,  
device,  
callbacks  
):  
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  
  
# 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()))  
  
# Save run settings  
if not evolve:  
with open(save\_dir / 'hyp.yaml', 'w') as f:  
yaml.safe\_dump(hyp, f, sort\_keys=False)  
with open(save\_dir / 'opt.yaml', 'w') as f:  
yaml.safe\_dump(vars(opt), f, sort\_keys=False)  
  
# Loggers  
data\_dict = None  
if RANK in [-1, 0]:  
loggers = Loggers(save\_dir, weights, opt, hyp, LOGGER) # loggers instance  
if loggers.wandb:  
data\_dict = loggers.wandb.data\_dict  
if resume:  
weights, epochs, hyp = opt.weights, opt.epochs, opt.hyp  
  
# Register actions  
for k in methods(loggers):  
callbacks.register\_action(k, callback=getattr(loggers, k))  
  
# Config  
plots = not evolve # create plots  
cuda = device.type != 'cpu'  
init\_seeds(1 + RANK)  
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 = ['item'] if single\_cls and len(data\_dict['names']) != 1 else data\_dict['names'] # class names  
assert len(names) == nc, f'{len(names)} names found for nc={nc} dataset in {data}' # check  
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=device) # load checkpoint  
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  
  
# Freeze  
freeze = [f'model.{x}.' for x in range(freeze)] # layers to freeze  
for k, v in model.named\_parameters():  
v.requires\_grad = True # train all layers  
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)  
  
# 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  
LOGGER.info(f"Scaled weight\_decay = {hyp['weight\_decay']}")  
g0, g1, g2 = [], [], [] # optimizer parameter groups  
for v in model.modules():  
if hasattr(v, 'bias') and isinstance(v.bias, nn.Parameter): # bias  
g2.append(v.bias)  
if isinstance(v, nn.BatchNorm2d): # weight (no decay)  
g0.append(v.weight)  
elif hasattr(v, 'weight') and isinstance(v.weight, nn.Parameter): # weight (with decay)  
g1.append(v.weight)  
if opt.adam:  
optimizer = Adam(g0, lr=hyp['lr0'], betas=(hyp['momentum'], 0.999)) # adjust beta1 to momentum  
else:  
optimizer = SGD(g0, lr=hyp['lr0'], momentum=hyp['momentum'], nesterov=True)  
optimizer.add\_param\_group({'params': g1, 'weight\_decay': hyp['weight\_decay']}) # add g1 with weight\_decay  
optimizer.add\_param\_group({'params': g2}) # add g2 (biases)  
LOGGER.info(f"{colorstr('optimizer:')} {type(optimizer).\_\_name\_\_} with parameter groups "  
f"{len(g0)} weight, {len(g1)} weight (no decay), {len(g2)} bias")  
del g0, g1, g2  
  
# Scheduler  
if opt.linear\_lr:  
lf = lambda x: (1 - x / (epochs - 1)) \* (1.0 - hyp['lrf']) + hyp['lrf'] # linear  
else:  
lf = one\_cycle(1, hyp['lrf'], epochs) # cosine 1->hyp['lrf']  
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  
start\_epoch, best\_fitness = 0, 0.0  
if pretrained:  
# Optimizer  
if ckpt['optimizer'] is not None:  
optimizer.load\_state\_dict(ckpt['optimizer'])  
best\_fitness = ckpt['best\_fitness']  
  
# EMA  
if ema and ckpt.get('ema'):  
ema.ema.load\_state\_dict(ckpt['ema'].float().state\_dict())  
ema.updates = ckpt['updates']  
  
# Epochs  
start\_epoch = ckpt['epoch'] + 1  
if resume:  
assert start\_epoch > 0, f'{weights} training to {epochs} epochs is finished, nothing to resume.'  
if epochs < start\_epoch:  
LOGGER.info(f"{weights} has been trained for {ckpt['epoch']} epochs. Fine-tuning for {epochs} more epochs.")  
epochs += ckpt['epoch'] # finetune additional epochs  
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://github.com/ultralytics/yolov5/issues/475 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=opt.cache, rect=opt.rect, rank=LOCAL\_RANK,  
workers=workers, image\_weights=opt.image\_weights, quad=opt.quad,  
prefix=colorstr('train: '), shuffle=True)  
mlc = int(np.concatenate(dataset.labels, 0)[:, 0].max()) # max label class  
nb = len(train\_loader) # number of batches  
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, pad=0.5,  
prefix=colorstr('val: '))[0]  
  
if not resume:  
labels = np.concatenate(dataset.labels, 0)  
# c = torch.tensor(labels[:, 0]) # classes  
# cf = torch.bincount(c.long(), minlength=nc) + 1. # frequency  
# model.\_initialize\_biases(cf.to(device))  
if plots:  
plot\_labels(labels, names, save\_dir)  
  
# Anchors  
if not opt.noautoanchor:  
check\_anchors(dataset, model=model, thr=hyp['anchor\_t'], imgsz=imgsz)  
model.half().float() # pre-reduce anchor precision  
  
callbacks.run('on\_pretrain\_routine\_end')  
  
# DDP mode  
if cuda and RANK != -1:  
model = DDP(model, device\_ids=[LOCAL\_RANK], output\_device=LOCAL\_RANK)  
  
# 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()  
nw = max(round(hyp['warmup\_epochs'] \* nb), 1000) # number of warmup iterations, max(3 epochs, 1k 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 = amp.GradScaler(enabled=cuda)  
stopper = EarlyStopping(patience=opt.patience)  
compute\_loss = ComputeLoss(model) # init loss class  
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 ------------------------------------------------------------------  
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  
  
mloss = torch.zeros(3, device=device) # mean losses  
if RANK != -1:  
train\_loader.sampler.set\_epoch(epoch)  
pbar = enumerate(train\_loader)  
LOGGER.info(('\n' + '%10s' \* 7) % ('Epoch', 'gpu\_mem', 'box', 'obj', 'cls', 'labels', 'img\_size'))  
if RANK in [-1, 0]:  
pbar = tqdm(pbar, total=nb, bar\_format='{l\_bar}{bar:10}{r\_bar}{bar:-10b}') # progress bar  
optimizer.zero\_grad()  
for i, (imgs, targets, paths, \_) in pbar: # batch -------------------------------------------------------------  
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 == 2 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(imgsz \* 0.5, 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 amp.autocast(enabled=cuda):  
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  
if ni - last\_opt\_step >= accumulate:  
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(('%10s' \* 2 + '%10.4g' \* 5) % (  
f'{epoch}/{epochs - 1}', mem, \*mloss, targets.shape[0], imgs.shape[-1]))  
callbacks.run('on\_train\_batch\_end', ni, model, imgs, targets, paths, plots, opt.sync\_bn)  
# 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, \_ = val.run(data\_dict,  
batch\_size=batch\_size // WORLD\_SIZE \* 2,  
imgsz=imgsz,  
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]  
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(),  
'wandb\_id': loggers.wandb.wandb\_run.id if loggers.wandb else None,  
'date': datetime.now().isoformat()}  
  
# Save last, best and delete  
torch.save(ckpt, last)  
if best\_fitness == fi:  
torch.save(ckpt, best)  
if (epoch > 0) and (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)  
  
# Stop Single-GPU  
if RANK == -1 and stopper(epoch=epoch, fitness=fi):  
break  
  
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, \_, \_ = val.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 results at 0.65  
single\_cls=single\_cls,  
dataloader=val\_loader,  
save\_dir=save\_dir,  
save\_json=is\_coco,  
verbose=True,  
plots=True,  
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, plots, epoch, results)  
LOGGER.info(f"Results saved to {colorstr('bold', save\_dir)}")  
torch.cuda.empty\_cache()  
return results  
  
def main(opt, callbacks=Callbacks()):  
# Checks  
if RANK in [-1, 0]:  
print\_args(FILE.stem, opt)  
check\_git\_status()  
check\_requirements(exclude=['thop'])  
  
# Resume  
if opt.resume and not check\_wandb\_resume(opt) and not opt.evolve: # resume an interrupted run  
ckpt = opt.resume if isinstance(opt.resume, str) else get\_latest\_run() # specified or most recent path  
assert os.path.isfile(ckpt), 'ERROR: --resume checkpoint does not exist'  
with open(Path(ckpt).parent.parent / 'opt.yaml', errors='ignore') as f:  
opt = argparse.Namespace(\*\*yaml.safe\_load(f)) # replace  
opt.cfg, opt.weights, opt.resume = '', ckpt, True # reinstate  
LOGGER.info(f'Resuming training from {ckpt}')  
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:  
opt.project = str(ROOT / 'runs/evolve')  
opt.exist\_ok, opt.resume = opt.resume, False # pass resume to exist\_ok and disable resume  
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:  
assert torch.cuda.device\_count() > LOCAL\_RANK, 'insufficient CUDA devices for DDP command'  
assert opt.batch\_size % WORLD\_SIZE == 0, '--batch-size must be multiple of CUDA device count'  
assert not opt.image\_weights, '--image-weights argument is not compatible with DDP training'  
assert not opt.evolve, '--evolve argument is not compatible with DDP training'  
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)  
if WORLD\_SIZE > 1 and RANK == 0:  
LOGGER.info('Destroying process group... ')  
dist.destroy\_process\_group()  
  
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)  
  
if \_\_name\_\_ == "\_\_main\_\_":  
opt = parse\_opt()  
main(opt)

1. detect.py

import argparse  
import os  
import sys  
from pathlib import Path  
import cv2  
import torch  
import torch.backends.cudnn as cudnn  
  
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  
  
from models.common import DetectMultiBackend  
from utils.datasets import IMG\_FORMATS, VID\_FORMATS, LoadImages, LoadStreams  
from utils.general import (LOGGER, check\_file, check\_img\_size, check\_imshow, check\_requirements, colorstr,  
increment\_path, non\_max\_suppression, print\_args, scale\_coords, strip\_optimizer, xyxy2xywh)  
from utils.plots import Annotator, colors, save\_one\_box  
from utils.torch\_utils import select\_device, time\_sync  
  
@torch.no\_grad()  
def run(weights=ROOT / 'yolov5s.pt', # model.pt path(s)  
source=ROOT / 'data/images', # file/dir/URL/glob, 0 for webcam  
imgsz=640, # inference size (pixels)  
conf\_thres=0.25, # confidence threshold  
iou\_thres=0.45, # NMS IOU threshold  
max\_det=1000, # maximum detections per image  
device='', # cuda device, i.e. 0 or 0,1,2,3 or cpu  
view\_img=False, # show results  
save\_txt=False, # save results to \*.txt  
save\_conf=False, # save confidences in --save-txt labels  
save\_crop=False, # save cropped prediction boxes  
nosave=False, # do not save images/videos  
classes=None, # filter by class: --class 0, or --class 0 2 3  
agnostic\_nms=False, # class-agnostic NMS  
augment=False, # augmented inference  
visualize=False, # visualize features  
update=False, # update all models  
project=ROOT / 'runs/detect', # save results to project/name  
name='exp', # save results to project/name  
exist\_ok=False, # existing project/name ok, do not increment  
line\_thickness=3, # bounding box thickness (pixels)  
hide\_labels=False, # hide labels  
hide\_conf=False, # hide confidences  
half=False, # use FP16 half-precision inference  
dnn=False, # use OpenCV DNN for ONNX inference  
):  
source = str(source)  
save\_img = not nosave and not source.endswith('.txt') # save inference images  
is\_file = Path(source).suffix[1:] in (IMG\_FORMATS + VID\_FORMATS)  
is\_url = source.lower().startswith(('rtsp://', 'rtmp://', 'http://', 'https://'))  
webcam = source.isnumeric() or source.endswith('.txt') or (is\_url and not is\_file)  
if is\_url and is\_file:  
source = check\_file(source) # download  
  
# Directories  
save\_dir = increment\_path(Path(project) / name, exist\_ok=exist\_ok) # increment run  
(save\_dir / 'labels' if save\_txt else save\_dir).mkdir(parents=True, exist\_ok=True) # make dir  
  
# Load model  
device = select\_device(device)  
model = DetectMultiBackend(weights, device=device, dnn=dnn)  
stride, names, pt, jit, onnx, engine = model.stride, model.names, model.pt, model.jit, model.onnx, model.engine  
imgsz = check\_img\_size(imgsz, s=stride) # check image size  
  
# Half  
half &= (pt or engine) and device.type != 'cpu' # half precision only supported by PyTorch on CUDA  
if pt:  
model.model.half() if half else model.model.float()  
  
# Dataloader  
if webcam:  
view\_img = check\_imshow()  
cudnn.benchmark = True # set True to speed up constant image size inference  
dataset = LoadStreams(source, img\_size=imgsz, stride=stride, auto=pt and not jit)  
bs = len(dataset) # batch\_size  
else:  
dataset = LoadImages(source, img\_size=imgsz, stride=stride, auto=pt and not jit)  
bs = 1 # batch\_size  
vid\_path, vid\_writer = [None] \* bs, [None] \* bs  
  
# Run inference  
model.warmup(imgsz=(1, 3, \*imgsz), half=half) # warmup  
dt, seen = [0.0, 0.0, 0.0], 0  
for path, im, im0s, vid\_cap, s in dataset:  
t1 = time\_sync()  
im = torch.from\_numpy(im).to(device)  
im = im.half() if half else im.float() # uint8 to fp16/32  
im /= 255 # 0 - 255 to 0.0 - 1.0  
if len(im.shape) == 3:  
im = im[None] # expand for batch dim  
t2 = time\_sync()  
dt[0] += t2 - t1  
  
# Inference  
visualize = increment\_path(save\_dir / Path(path).stem, mkdir=True) if visualize else False  
pred = model(im, augment=augment, visualize=visualize)  
t3 = time\_sync()  
dt[1] += t3 - t2  
  
# NMS  
pred = non\_max\_suppression(pred, conf\_thres, iou\_thres, classes, agnostic\_nms, max\_det=max\_det)  
dt[2] += time\_sync() - t3  
  
# Second-stage classifier (optional)  
# pred = utils.general.apply\_classifier(pred, classifier\_model, im, im0s)  
  
# Process predictions  
for i, det in enumerate(pred): # per image  
seen += 1  
if webcam: # batch\_size >= 1  
p, im0, frame = path[i], im0s[i].copy(), dataset.count  
s += f'{i}: '  
else:  
p, im0, frame = path, im0s.copy(), getattr(dataset, 'frame', 0)  
p = Path(p) # to Path  
save\_path = str(save\_dir / p.name) # im.jpg  
txt\_path = str(save\_dir / 'labels' / p.stem) + ('' if dataset.mode == 'image' else f'\_{frame}') # im.txt  
#s += '%gx%g ' % im.shape[2:] # print string  
gn = torch.tensor(im0.shape)[[1, 0, 1, 0]] # normalization gain whwh  
imc = im0.copy() if save\_crop else im0 # for save\_crop  
annotator = Annotator(im0, line\_width=line\_thickness, example=str(names))  
if len(det):  
# Rescale boxes from img\_size to im0 size  
det[:, :4] = scale\_coords(im.shape[2:], det[:, :4], im0.shape).round()  
  
# Print results  
for c in det[:, -1].unique():  
n = (det[:, -1] == c).sum() # detections per class  
# s += f"{n} {names[int(c)]}{'s' \* (n > 1)}, " # add to string  
  
# Write results  
for \*xyxy, conf, cls in reversed(det):  
if save\_txt: # Write to file  
xywh = (xyxy2xywh(torch.tensor(xyxy).view(1, 4)) / gn).view(-1).tolist() # normalized xywh  
line = (cls, \*xywh, conf) if save\_conf else (cls, \*xywh) # label format  
with open(txt\_path + '.txt', 'a') as f:  
f.write(('%g ' \* len(line)).rstrip() % line + '\n')  
if save\_img or save\_crop or view\_img: # Add bbox to image  
c = int(cls) # integer class  
label = None if hide\_labels else (names[c] if hide\_conf else f'{names[c]} {conf:.2f}')  
annotator.box\_label(xyxy, label, color=colors(c, True))  
if save\_crop:  
save\_one\_box(xyxy, imc, file=save\_dir / 'crops' / names[c] / f'{p.stem}.jpg', BGR=True)  
  
# Print time (inference-only)  
LOGGER.info(f'{s}Done. ({t3 - t2:.3f}s)')  
  
# Stream results  
im0 = annotator.result()  
if view\_img:  
cv2.imshow(str(p), im0)  
cv2.waitKey(1) # 1 millisecond  
  
# Save results (image with detections)  
if save\_img:  
if dataset.mode == 'image':  
cv2.imwrite(save\_path, im0)  
else: # 'video' or 'stream'  
if vid\_path[i] != save\_path: # new video  
vid\_path[i] = save\_path  
if isinstance(vid\_writer[i], cv2.VideoWriter):  
vid\_writer[i].release() # release previous video writer  
if vid\_cap: # video  
fps = vid\_cap.get(cv2.CAP\_PROP\_FPS)  
w = int(vid\_cap.get(cv2.CAP\_PROP\_FRAME\_WIDTH))  
h = int(vid\_cap.get(cv2.CAP\_PROP\_FRAME\_HEIGHT))  
else: # stream  
fps, w, h = 30, im0.shape[1], im0.shape[0]  
save\_path += '.mp4'  
vid\_writer[i] = cv2.VideoWriter(save\_path, cv2.VideoWriter\_fourcc(\*'mp4v'), fps, (w, h))  
vid\_writer[i].write(im0)  
  
# Print results  
t = tuple(x / seen \* 1E3 for x in dt) # speeds per image  
LOGGER.info(f'Speed: %.1fms pre-process, %.1fms inference, %.1fms NMS per image at shape {(1, 3, \*imgsz)}' % t)  
if save\_txt or save\_img:  
s = f"\n{len(list(save\_dir.glob('labels/\*.txt')))} labels saved to {save\_dir / 'labels'}" if save\_txt else ''  
LOGGER.info(f"Results saved to {colorstr('bold', save\_dir)}{s}")  
if update:  
strip\_optimizer(weights) # update model (to fix SourceChangeWarning)  
  
def main(opt):  
check\_requirements(exclude=('tensorboard', 'thop'))  
run(\*\*vars(opt))  
  
if \_\_name\_\_ == "\_\_main\_\_":  
opt = parse\_opt()  
main(opt)

1. coco.yaml

path: E:/training/yolov5 # dataset root dir  
train: E:/training/yolov5/images # train images (relative to 'path')

val: E:/training/yolov5/images # train images (relative to 'path')

test: E:/training/yolov5/images # 20288 of 40670 images, submit to https://competitions.codalab.org/competitions/20794  
  
# Classes  
nc: 2 # number of classes  
names: ['face', 'body'] # class names

1. coco.names

face  
body