AERIAL VIEW OBJECT CLASSIFICATION ON THE BIRDSAI DATASET

Colin Hanley
cjhanley@crimson.ua.edu

Jasmine Pierre
jppierre@crimson.ua.edu

Gowtham Prasad
gdprasad@crimson.ua.edu

May 1, 2025

ABSTRACT

With increased poacher activity targeting rhinos and elephants, conservation organizations are left responsible for monitoring large swaths of land for unauthorized human presence. The current system for surveillance of this land is manual inspection of long-wave infrared (LWIR) video feed captured by UAVs. This solution, while functional, is tedious and is subject to human error. We propose a real-time computer vision monitoring system based on a pre-trained YOLOv5 model to distinguish between humans and animals in LWIR imagery. By combining pseudo-labeling of synthetic data with weighted cross-entropy loss, our approach was able to achieve 76% recall on the human class, a steep improvement over the 39% achieved from training on just real images.

1 Background/Theory

A similar project was conducted by Bondi et al. [1] in 2018. Their project utilized an RCNN model, opting for the improved accuracy of this model over other potential architectures. In field testing, their model was able to achieve approximately 37% recall in the human class, which was a significant improvement over the current human monitoring system. We hope to build off of this project using a YOLO architecture for increased inference speed.
The YOLO architecture we use for our model was first described by Redmon et al.[2] in 2015. These models frame object detection problems as a regression problem, allowing for inference in a single-pass of the image. This speeds up the model, which is critical for our model to perform real-time analysis of LWIR video feed.
The YOLOv5 model specifically is constructed with a CSPDarkNet-53 backbone, which is a 53-layer convolutional neural network. CSPNets were first introduced by Wang et al[3]. CSPNet splits the input feature map x_0 into two parts:

$$x_0 = [x_0^{'}, x_0^{''}]$$

- Path 1 ($x_0^{''}$) passes through the network
- Path 2 ($x_0^{'}$) bypasses the network and is fused afterwards

This increases gradient diversity, as well as reduces the computational load of training the model. YOLOv5 models also make use of anchor boxes described by Zhong et al[4], which are a set of hypothetical bounding boxes produced during training that represent the different sizes of objects present in the training set. This allows the model to perform a refinement problem at inference time rather than forming new bounding boxes for each image, increasing the accuracy of the model.

2 Data

The BIRDSAI [5] dataset is composed of 27000 LWIR images of the African savannah, of which only 447 are real and labeled. The remainder are synthetically generated and unlabeled.

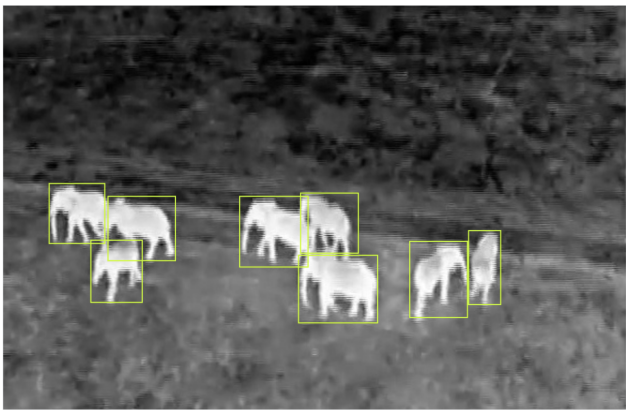


Figure 1: An example of the real, labeled data provided

In order to generate more training data, we applied an initial YOLOv5 detector that was trained exlcusively on the 447 real images to a randomly selected subset of 553 synthetic images. After manual verification of these pseudo-labels, we assembled a 1000 image labeled set. Our dataset has two issues that we encountered through our training process. For one, there is a severe class imbalance. In our newly generated 1000 image dataset, there are only 280 instances of the human class, as compared to 4503 instances of the animal class. This issue was somewhat addressed in the generation of our pseudo-labels, as the original dataset only had 140 human instances, but this imbalance still provides an issue in training our model. Secondly, the images in our dataset are taken from a variety of vantage points.

Figure 2: An example of an image in our dataset taken from a high vantage point

As can be clearly seen in comparing the two provided images, despite both images containing members of the animal class, the appearance, most importantly the size, of the instances look quite different depending on the vantage point. This difference requires our model to generalize well to the appearance of each class from different perspectives.

3 Methodology

We adpoted a pretrained YOLOv5 with a CSPDarknet-53 backbone. This choice was made for its favorable inference speed while still maintaining a good level of accuracy. We trained our model for 10 epochs.

3.1 Loss Function

Given the class imbalance, we applied a weighted cross-entropy loss defined as

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}[w_{human}y_i log p_i + w_{animal}(1-y_i)log(1-p_i)] \tag{1}$$

where w_{human} and w_{animal} are the weights assigned to each class. We set these weights to $w_{human} = 16$ and $w_{animal} = 1$, as the ratio between animal and human instances in the training set is 16:1

3.2 Optimizer

We trained the model using the Adam optimizer with momentum $= 0.937$ and weight decay = 5×10^{-5}

4 Results/Analysis

4.1 Initial Model

Our initial model was trained only on the 447 real images provided in the BIRDSAI dataset. This model was then applied to an unseen test set, which was made up of 200 hand-labeled images that were randomly selected from the synthetic data provided. This resulted in the confusion matrix below

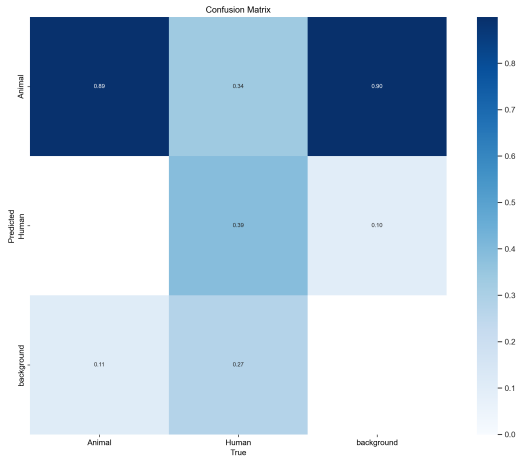


Figure 3: Confusion Matrix of initial YOLOv5 model

The metric that we are most concerned with is recall for the human class, as in the real-life application of this model, catching all instances of humans in the video footage is more important than preventing an occasional misclassification. As can be seen in the confusion matrix given, the initial model that we trained had relatively poor recall for the human class at 39%. However, this model was useful in generating pseudo-labels for the 553 synthetic images that we introduced into our training set

4.2 Model Trained with Pseudo-labeled Data

The YOLOv5 model was then re-trained with the same hyperparameters on this 1000 image dataset, which resulted in the confusion matrix below.

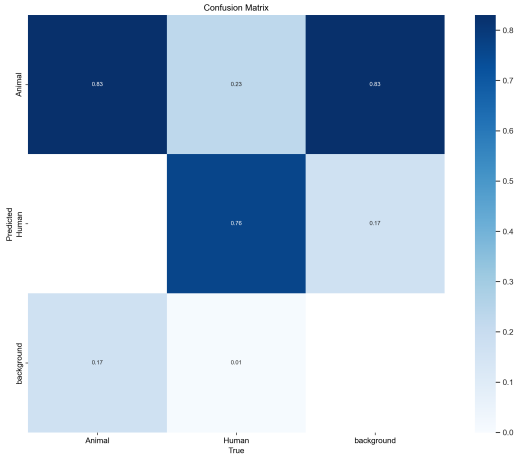
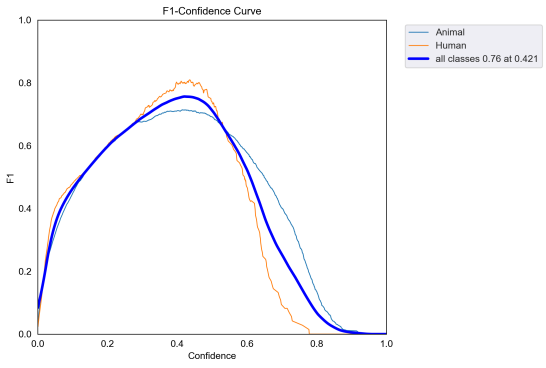
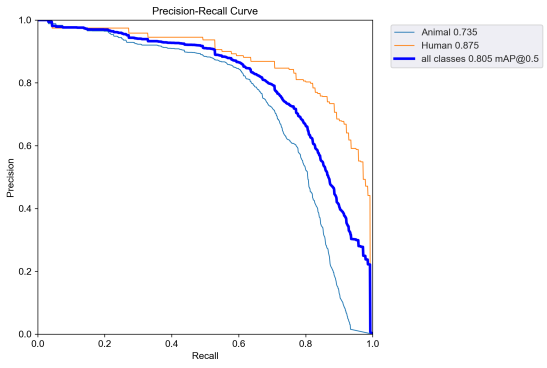


Figure 4: Confusion Matrix of retrained YOLOv5 model

As can be seen in this matrix, the recall of the human class is much improved when the model was retrained on the pseudo-labeled dataset, moving from 39% to 76%. Another note from the confusion matrix is the apparently common prediction of background(or our null class) as animals. We believe that this is a result of bounding boxes formed on inference overlapping with sections of background as well. This is likely due to the data's varied vantage points, which, as mentioned in the Data section, results in varied sizes of animal instances in the training set and therefore, difficulty in finding the correct bounding box size on inference. The vantage point of the images containing human instances seem to be more consistent across the dataset, which may explain why there is less occurrence of human predictions overlapping with the background. We also produced the following visualizations, demonstrating the performance of the model in predicting the two classes

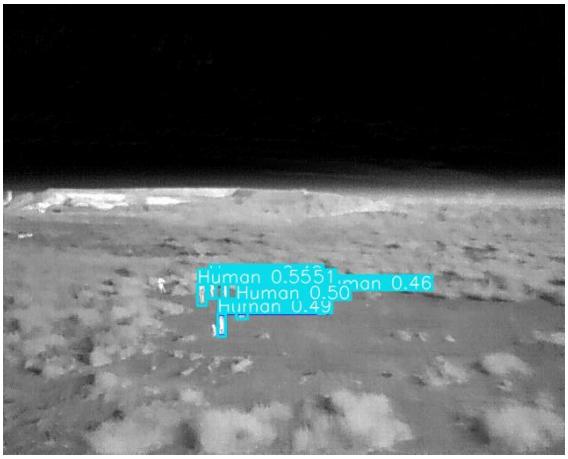


(a) F1-Confidence Curve

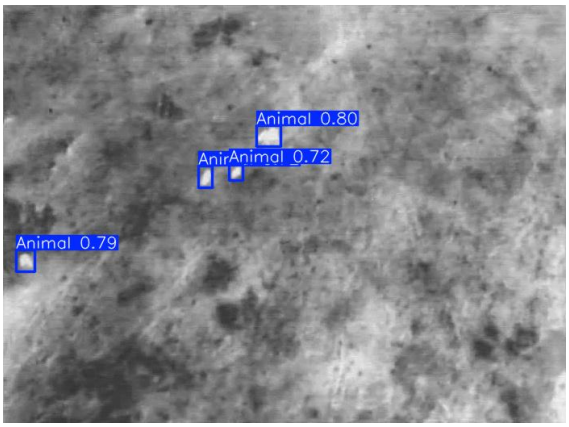


(b) Precision-Recall Curve

The output of this model looks like the images below



(a) Inference on Human Class



(b) Inference on Animal Class

As can be seen in these two images, the model often does not make high-confidence predictions on instances of the human class as a result of a lack of instances in the training set. It can also be seen that the model failed to identify a human on the left side of the image, but was able to identify a majority of the cluster.
Another key feature of the model is its ability to perform inference quickly, allowing for the real-time analysis of LWIR video feed.

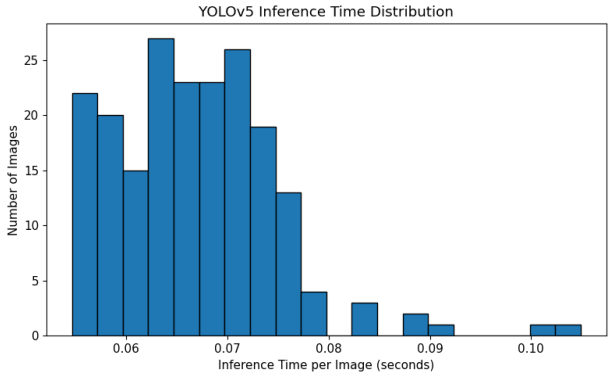


Figure 7: Inference Speed Distribution on Test Set(200 images)

As can be seen in the histogram above, a vast majority of inferences on a single video frame can be performed in less than .1 seconds, with an average inference time of .0669 seconds. This allows for the real-time analysis of 9 Hz LWIR video feed, which is a common frame-rate for this type of camera

5 Discussion

We have designed a computer vision model built off of a pre-trained YOLOv5 that is capable of identifying and differentiating between humans and animals in long-wave infrared images with fairly high success. Through the production of more labeled data and weighted loss, we were able to create a model that is capable of identifying humans at a relatively high rate despite a class imbalance in the dataset. This system has the potential to be applied to real-time LWIR video feed collected by UAVs, allowing for more efficient tracking of human activity in protected wildlife areas and, ideally, reduction in poaching rates.

References

[1] Elizabeth Bondi, Fei Fang, Mark Hamilton, Debarun Kar, Donnabell Dmello, Jongmoo Choi, Robert Hannaford, Arvind Iyer, Lucas Joppa, Milind Tambe, and Ram Nevatia. Spot poachers in action: Augmenting conservation

drones with automatic detection in near real time. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), Apr. 2018.

[2] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *CoRR*, abs/1506.02640, 2015.

[3] Chien-Yao Wang, Hong-Yuan Mark Liao, I-Hau Yeh, Yueh-Hua Wu, Ping-Yang Chen, and Jun-Wei Hsieh. Cspnet: A new backbone that can enhance learning capability of CNN. *CoRR*, abs/1911.11929, 2019.

[4] Yuanyi Zhong, Jianfeng Wang, Jian Peng, and Lei Zhang. Anchor box optimization for object detection. *CoRR*, abs/1812.00469, 2018.

[5] Elizabeth Bondi, Raghav Jain, Palash Aggrawal, Saket Anand, Robert Hannaford, Ashish Kapoor, Jim Piavis, Shital Shah, Lucas Joppa, Bistra Dilkina, and Milind Tambe. Birdsai: A dataset for detection and tracking in aerial thermal infrared videos. In *WACV*, 2020.

```
'''
Primary script to invoke train.py and fine-tune the model using the labeled images
'''
import subprocess
import os
import shutil
import yaml

def train_yolo(data_yaml, weights, epochs, batch_size, img_size, exp_name, dropoutrate):

    cmd = [
        "python3", "yolov5/train.py",
        "--data", data_yaml,
        "--weights", weights,
        "--epochs", str(epochs),
        "--batch-size", str(batch_size),
        "--img", str(img_size),
        "--name", exp_name,
        "--dropout", str(dropoutrate)
    ]
    subprocess.run(cmd, check=True)


def main():
    labeled_images_dir = "./labeled/images"
    labeled_labels_dir = "./labeled/labels"

    labeled_data_yaml = "./data.yaml"

    initial_epochs = 10
    retrain_epochs = 10
    batch_size = 16
    img_size = 640
    dropoutrate = 0.2
    confidence_threshold = 0.9
    num_classes = 2
    names = ["Animal", "Human"]

    initial_exp_name = "train"


    base_weights = "yolov5s.pt"

    train_yolo(
        labeled_data_yaml,
        base_weights,
        initial_epochs,
        batch_size,
        img_size,
        initial_exp_name,
        dropoutrate
    )

    weights_trained = f"runs/train/{initial_exp_name}/weights/best.pt"


if __name__ == '__main__':
    main()
```

```
'''
Script to perform inference using the trained model on an unseen test dataset
'''

import os
import time
import torch
import numpy as np
from pycocotools.coco import COCO
from pycocotools.cocoeval import COCOeval
from sklearn.metrics import confusion_matrix, roc_curve, auc
import matplotlib.pyplot as plt



def load_model(weights_path):

    return torch.hub.load('ultralytics/yolov5', 'custom', path=weights_path, force_reloa
d=True)


def load_ground_truth(gt_json_path):

    return COCO(gt_json_path)


def generate_detections(model, coco_gt, images_dir):

    detections = []
    inference_times = []
    obj_counts = []
    cat_ids = coco_gt.getCatIds()

    for img_id in coco_gt.getImgIds():
        info = coco_gt.loadImgs(img_id)[0]
        img_path = os.path.join(images_dir, info['file_name'])
        if not os.path.exists(img_path):
            print(f"Missing image: {img_path}")
            continue

        # --- timing start
        t0 = time.time()
        results = model(img_path)
        t1 = time.time()
        inference_times.append(t1 - t0)
        # --- timing end

        df = results.pandas().xyxy[0]
        obj_counts.append(len(df))

        for _, row in df.iterrows():
            detections.append({
                'image_id': img_id,
                'category_id': cat_ids[int(row['class'])],
                'bbox': [row['xmin'], row['ymin'],
                         row['xmax'] - row['xmin'],
                         row['ymax'] - row['ymin']],
                'score': float(row['confidence'])
            })
```

```
    return detections, inference_times, obj_counts


def plot_inference_speed(times, output_path='inference_speed.png'):

    plt.figure(figsize=(8, 5))
    plt.hist(times, bins=20, edgecolor='black')
    plt.xlabel('Inference Time per Image (seconds)')
    plt.ylabel('Number of Images')
    plt.title('YOLOv5 Inference Time Distribution')
    plt.tight_layout()
    plt.savefig(output_path)
    print(f'Saved inference speed histogram to {output_path}')
    plt.show()


def plot_speed_vs_count(times, counts, output_path='speed_vs_count.png'):

    plt.figure(figsize=(8, 5))
    plt.scatter(counts, times)
    plt.xlabel('Number of Detections')
    plt.ylabel('Inference Time (seconds)')
    plt.title('Inference Time vs. # Detections per Image')

    if counts:
        max_count = max(counts)
        ticks = np.arange(0, max_count + 1, 2)
        plt.xticks(ticks)

    plt.tight_layout()
    plt.savefig(output_path)
    print(f'Saved inference time vs. count plot to {output_path}')
    plt.show()


def evaluate_coco(coco_gt, detections, iou_thr=0.5):
    """
    Compute COCO mAP at a single IoU threshold.
    """
    coco_dt = coco_gt.loadRes(detections)
    coco_eval = COCOeval(coco_gt, coco_dt, 'bbox')
    coco_eval.params.iouThrs = np.array([iou_thr])
    coco_eval.evaluate()
    coco_eval.accumulate()
    coco_eval.summarize()


def compute_and_plot_metrics(detections, coco_gt, iou_thr=0.5):
    """
    Build and display ROC curve and a per-GT normalized confusion matrix.
    """
    gt_by_img = {}
    for ann in coco_gt.loadAnns(coco_gt.getAnnIds()):
        gt_by_img.setdefault(ann['image_id'], []).append((ann['bbox'], ann['category_id'
]))

    y_true, y_scores = [], []
    cm_true, cm_pred = [], []

    for det in detections:
```

```
    img_id, dbox, dcat = det['image_id'], det['bbox'], det['category_id']
    score = det['score']
    y_scores.append(score)
    is_tp = 0
    if img_id in gt_by_img:
        db = np.array([dbox[0], dbox[1], dbox[0]+dbox[2], dbox[1]+dbox[3]])
        best_iou, best_cat = 0, None
        for gbox, gcat in gt_by_img[img_id]:
            gb = np.array([gbox[0], gbox[1], gbox[0]+gbox[2], gbox[1]+gbox[3]])
            ixmin = max(gb[0], db[0]); iymin = max(gb[1], db[1])
            ixmax = min(gb[2], db[2]); iymax = min(gb[3], db[3])
            iw = max(ixmax-ixmin, 0); ih = max(iymax-iymin, 0)
            inter = iw * ih
            union = ((db[2]-db[0])*(db[3]-db[1]) +
                     (gb[2]-gb[0])*(gb[3]-gb[1]) - inter)
            iou = inter/union if union > 0 else 0
            if iou > best_iou:
                best_iou, best_cat = iou, gcat
        if best_iou >= iou_thr:
            is_tp = 1
            cm_true.append(best_cat)
            cm_pred.append(dcat)
    y_true.append(is_tp)

fpr, tpr, _ = roc_curve(y_true, y_scores)
roc_auc = auc(fpr, tpr)
plt.figure()
plt.plot(fpr, tpr, lw=2, label=f'AUC = {roc_auc:.2f}')
plt.plot([0,1], [0,1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Detection ROC Curve')
plt.legend(loc='lower right')
plt.tight_layout()
plt.savefig('roc_curve.png')
print('Saved ROC curve to roc_curve.png')
plt.show()

cat_ids = coco_gt.getCatIds()
cat_info = coco_gt.loadCats(cat_ids)
cat_names = [c['name'] for c in cat_info]
cm = confusion_matrix(cm_true, cm_pred, labels=cat_ids)
cm = cm.astype(float)
col_sums = cm.sum(axis=0, keepdims=True)
cm_norm = np.divide(cm, col_sums, where=col_sums>0)

fig, ax = plt.subplots(figsize=(6,6))
im = ax.imshow(cm_norm, interpolation='nearest', cmap=plt.cm.Blues)
fig.colorbar(im, ax=ax)

ax.set_xticks(np.arange(len(cat_names)))
ax.set_yticks(np.arange(len(cat_names)))
ax.set_xticklabels(cat_names, rotation=45, ha='right')
ax.set_yticklabels(cat_names)

for i in range(len(cat_names)):
    for j in range(len(cat_names)):
        val = cm_norm[i, j]
        color = 'white' if val > 0.5 else 'black'
        ax.text(j, i, f"{val:.2f}", ha='center', va='center', color=color)
```

```
    ax.set_xlabel('Ground Truth')
    ax.set_ylabel('Predicted')
    ax.set_title('Confusion Matrix (per-GT fractions)')
    plt.tight_layout()
    plt.savefig('confusion_matrix.png')
    print('Saved confusion matrix to confusion_matrix.png')
    plt.show()


def main():
    gt_json = os.path.join('testdata', '_annotations.coco.json')
    imgs_dir = os.path.join('testdata', 'images')
    weights = 'yolov5/runs/train/labeled_train/weights/best.pt'

    model = load_model(weights)
    coco_gt = load_ground_truth(gt_json)

    dets, times, counts = generate_detections(model, coco_gt, imgs_dir)

    avg_time = np.mean(times) if times else float('nan')
    print(f'Average inference time per image: {avg_time:.4f} seconds')

    plot_inference_speed(times)

    plot_speed_vs_count(times, counts)

    evaluate_coco(coco_gt, dets)
    compute_and_plot_metrics(dets, coco_gt)


if __name__ == '__main__':
    main()
```

```
'''
Base YOLOv5 training script

'''

import argparse
import math
import os
import random
import subprocess
import sys
import time
from copy import deepcopy
from datetime import datetime, timedelta
from pathlib import Path


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]
if str(ROOT) not in sys.path:
    sys.path.append(str(ROOT))
ROOT = Path(os.path.relpath(ROOT, Path.cwd()))

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, 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))
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):

    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.ite
ms()))
    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}:
        include_loggers = list(LOGGERS)
        if getattr(opt, "ndjson_console", False):
            include_loggers.append("ndjson_console")
        if getattr(opt, "ndjson_file", False):
            include_loggers.append("ndjson_file")

        loggers = Loggers(
            save_dir=save_dir,
            weights=weights,
            opt=opt,
            hyp=hyp,
            logger=LOGGER,
            include=tuple(include_loggers),
        )

        # 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.bat
ch_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["n
ames"]  # class names
    is_coco = isinstance(val_path, str) and val_path.endswith("coco/val2017.txt")  # COC
O 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 avoi
d 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 []  # ex
clude 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 {weigh
ts}")  # 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 erra
tic 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:

        def lf(x):
            """Linear learning rate scheduler function with decay calculated by epoch pr
oportion."""
            return (1 - x / epochs) * (1.0 - hyp["lrf"]) + hyp["lrf"]  # linear

    scheduler = lr_scheduler.LambdaLR(optimizer, lr_lambda=lf)  # plot_lr_scheduler(opti
mizer, 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, weigh
ts, 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/mul
ti_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 label
s 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 weigh
ts
            iw = labels_to_image_weights(dataset.labels, nc=nc, class_weights=cw)  # ima
ge 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 * g
s  # 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 sh
ape (stretched to gs-multiple)
                    imgs = nn.functional.interpolate(imgs, size=ns, mode="bilinear", ali
gn_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 *= 4.0

            # 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)  # cli
p 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.shap
e[-1])
                )
                callbacks.run("on_train_batch_end", model, ni, imgs, targets, paths, lis
t(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", "cla
ss_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, f
i)

        # 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 ran
ks
            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):
    """
    Parse command-line arguments for YOLOv5 training, validation, and testing.

    Args:
        known (bool, optional): If True, parses known arguments, ignoring the unknown. D
efaults to False.

    Returns:
        (argparse.Namespace): Parsed command-line arguments containing options for YOLOv
5 execution.

    Example:
        ```python
        from ultralytics.yolo import parse_opt
        opt = parse_opt()
        print(opt)
        ```


    Links:
        - 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
    """
    parser = argparse.ArgumentParser()
    parser.add_argument("--weights", type=str, default=ROOT / "yolov5s.pt", help="initia
l weights path")
    parser.add_argument("--cfg", type=str, default="", help="model.yaml path")
    parser.add_argument("--data", type=str, default=ROOT / "data/coco128.yaml", help="da
taset.yaml path")
    parser.add_argument("--hyp", type=str, default=ROOT / "data/hyps/hyp.scratch-low.yam
l", help="hyperparameters path")
    parser.add_argument("--epochs", type=int, default=100, help="total training epochs")
    parser.add_argument("--batch-size", type=int, default=16, help="total batch size for
 all GPUs, -1 for autobatch")
    parser.add_argument("--imgsz", "--img", "--img-size", type=int, default=640, help="t
rain, 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 m
ost recent training")
    parser.add_argument("--nosave", action="store_true", help="only save final checkpoin
t")
    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 hyperpa
rameters for x generations")
    parser.add_argument(
        "--evolve_population", type=str, default=ROOT / "data/hyps", help="location for
loading population"
)
    parser.add_argument("--resume_evolve", type=str, default=None, help="resume evolve f
rom last generation")
```

```
    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="", 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 dat
a as single-class")
    parser.add_argument("--optimizer", type=str, choices=["SGD", "Adam", "AdamW"], defau
lt="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=8, 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 o
k, 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 smooth
ing epsilon")
    parser.add_argument("--patience", type=int, default=100, help="EarlyStopping patienc
e (epochs without improvement)")
    parser.add_argument("--freeze", nargs="+", type=int, default=[0], help="Freeze layer
s: backbone=10, first3=0 1 2")
    parser.add_argument("--save-period", type=int, default=-1, help="Save checkpoint eve
ry 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")

    # NDJSON logging
    parser.add_argument("--ndjson-console", action="store_true", help="Log ndjson to con
sole")
    parser.add_argument("--ndjson-file", action="store_true", help="Log ndjson to file")

    return parser.parse_known_args()[0] if known else parser.parse_args()


def main(opt, callbacks=Callbacks()):
    """
    Runs the main entry point for training or hyperparameter evolution with specified op
tions and optional callbacks.

    Args:
        opt (argparse.Namespace): The command-line arguments parsed for YOLOv5 training
```

```
and evolution.
        callbacks (ultralytics.utils.callbacks.Callbacks, optional): Callback functions
for various training stages.
            Defaults to Callbacks().

    Returns:
        None

    Note:
        For detailed usage, refer to:
        https://github.com/ultralytics/yolov5/tree/master/models
    """
    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_lates
t_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 spec
ified"
        if opt.evolve:
            if opt.project == str(ROOT / "runs/train"):  # if default project name, rena
me 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.exi
st_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 DD
P 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", timeout=timedelta(se
conds=10800)
        )

    # Train
    if not opt.evolve:
        train(opt.hyp, opt, device, callbacks)

    # Evolve hyperparameters (optional)
    else:
        # Hyperparameter evolution metadata (including this hyperparameter True-False, l
ower_limit, upper_limit)
        meta = {
            "lr0": (False, 1e-5, 1e-1),  # initial learning rate (SGD=1E-2, Adam=1E-3)
            "lrf": (False, 0.01, 1.0),  # final OneCycleLR learning rate (lr0 * lrf)
            "momentum": (False, 0.6, 0.98),  # SGD momentum/Adam beta1
            "weight_decay": (False, 0.0, 0.001),  # optimizer weight decay
            "warmup_epochs": (False, 0.0, 5.0),  # warmup epochs (fractions ok)
            "warmup_momentum": (False, 0.0, 0.95),  # warmup initial momentum
            "warmup_bias_lr": (False, 0.0, 0.2),  # warmup initial bias lr
            "box": (False, 0.02, 0.2),  # box loss gain
            "cls": (False, 0.2, 4.0),  # cls loss gain
            "cls_pw": (False, 0.5, 2.0),  # cls BCELoss positive_weight
            "obj": (False, 0.2, 4.0),  # obj loss gain (scale with pixels)
            "obj_pw": (False, 0.5, 2.0),  # obj BCELoss positive_weight
            "iou_t": (False, 0.1, 0.7),  # IoU training threshold
            "anchor_t": (False, 2.0, 8.0),  # anchor-multiple threshold
            "anchors": (False, 2.0, 10.0),  # anchors per output grid (0 to ignore)
            "fl_gamma": (False, 0.0, 2.0),  # focal loss gamma (efficientDet default gam
ma=1.5)
            "hsv_h": (True, 0.0, 0.1),  # image HSV-Hue augmentation (fraction)
            "hsv_s": (True, 0.0, 0.9),  # image HSV-Saturation augmentation (fraction)
            "hsv_v": (True, 0.0, 0.9),  # image HSV-Value augmentation (fraction)
            "degrees": (True, 0.0, 45.0),  # image rotation (+/- deg)
            "translate": (True, 0.0, 0.9),  # image translation (+/- fraction)
            "scale": (True, 0.0, 0.9),  # image scale (+/- gain)
            "shear": (True, 0.0, 10.0),  # image shear (+/- deg)
            "perspective": (True, 0.0, 0.001),  # image perspective (+/- fraction), rang
e 0-0.001
            "flipud": (True, 0.0, 1.0),  # image flip up-down (probability)
            "fliplr": (True, 0.0, 1.0),  # image flip left-right (probability)
            "mosaic": (True, 0.0, 1.0),  # image mosaic (probability)
            "mixup": (True, 0.0, 1.0),  # image mixup (probability)
            "copy_paste": (True, 0.0, 1.0),  # segment copy-paste (probability)
        }

        # GA configs
        pop_size = 50
        mutation_rate_min = 0.01
        mutation_rate_max = 0.5
        crossover_rate_min = 0.5
        crossover_rate_max = 1
```

```
        min_elite_size = 2
        max_elite_size = 5
        tournament_size_min = 2
        tournament_size_max = 10

        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/sav
e 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),
                ]
            )

        # Delete the items in meta dictionary whose first value is False
        del_ = [item for item, value_ in meta.items() if value_[0] is False]
        hyp_GA = hyp.copy()  # Make a copy of hyp dictionary
        for item in del_:
            del meta[item]  # Remove the item from meta dictionary
            del hyp_GA[item]  # Remove the item from hyp_GA dictionary

        # Set lower_limit and upper_limit arrays to hold the search space boundaries
        lower_limit = np.array([meta[k][1] for k in hyp_GA.keys()])
        upper_limit = np.array([meta[k][2] for k in hyp_GA.keys()])

        # Create gene_ranges list to hold the range of values for each gene in the popul
ation
        gene_ranges = [(lower_limit[i], upper_limit[i]) for i in range(len(upper_limit))
]

        # Initialize the population with initial_values or random values
        initial_values = []

        # If resuming evolution from a previous checkpoint
        if opt.resume_evolve is not None:
            assert os.path.isfile(ROOT / opt.resume_evolve), "evolve population path is
wrong!"
            with open(ROOT / opt.resume_evolve, errors="ignore") as f:
                evolve_population = yaml.safe_load(f)
                for value in evolve_population.values():
                    value = np.array([value[k] for k in hyp_GA.keys()])
                    initial_values.append(list(value))

        # If not resuming from a previous checkpoint, generate initial values from .yaml
 files in opt.evolve_population
        else:
            yaml_files = [f for f in os.listdir(opt.evolve_population) if f.endswith(".y
aml")]
```

```
            for file_name in yaml_files:
                with open(os.path.join(opt.evolve_population, file_name)) as yaml_file:
                    value = yaml.safe_load(yaml_file)
                    value = np.array([value[k] for k in hyp_GA.keys()])
                    initial_values.append(list(value))

        # Generate random values within the search space for the rest of the population
        if initial_values is None:
            population = [generate_individual(gene_ranges, len(hyp_GA)) for _ in range(p
op_size)]
        elif pop_size > 1:
            population = [generate_individual(gene_ranges, len(hyp_GA)) for _ in range(p
op_size - len(initial_values))]
            for initial_value in initial_values:
                population = [initial_value] + population

        # Run the genetic algorithm for a fixed number of generations
        list_keys = list(hyp_GA.keys())
        for generation in range(opt.evolve):
            if generation >= 1:
                save_dict = {}
                for i in range(len(population)):
                    little_dict = {list_keys[j]: float(population[i][j]) for j in range(
len(population[i]))}
                    save_dict[f"gen{str(generation)}number{str(i)}"] = little_dict

                with open(save_dir / "evolve_population.yaml", "w") as outfile:
                    yaml.dump(save_dict, outfile, default_flow_style=False)

            # Adaptive elite size
            elite_size = min_elite_size + int((max_elite_size - min_elite_size) * (gener
ation / opt.evolve))
            # Evaluate the fitness of each individual in the population
            fitness_scores = []
            for individual in population:
                for key, value in zip(hyp_GA.keys(), individual):
                    hyp_GA[key] = value
                hyp.update(hyp_GA)
                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)
                fitness_scores.append(results[2])

            # Select the fittest individuals for reproduction using adaptive tournament
selection
            selected_indices = []
            for _ in range(pop_size - elite_size):
                # Adaptive tournament size
                tournament_size = max(
                    max(2, tournament_size_min),
```

```
                    int(min(tournament_size_max, pop_size) - (generation / (opt.evolve /
 10))),
                )
                # Perform tournament selection to choose the best individual
                tournament_indices = random.sample(range(pop_size), tournament_size)
                tournament_fitness = [fitness_scores[j] for j in tournament_indices]
                winner_index = tournament_indices[tournament_fitness.index(max(tournamen
t_fitness))]
                selected_indices.append(winner_index)

            # Add the elite individuals to the selected indices
            elite_indices = [i for i in range(pop_size) if fitness_scores[i] in sorted(f
itness_scores)[-elite_size:]]
            selected_indices.extend(elite_indices)
            # Create the next generation through crossover and mutation
            next_generation = []
            for _ in range(pop_size):
                parent1_index = selected_indices[random.randint(0, pop_size - 1)]
                parent2_index = selected_indices[random.randint(0, pop_size - 1)]
                # Adaptive crossover rate
                crossover_rate = max(
                    crossover_rate_min, min(crossover_rate_max, crossover_rate_max - (ge
neration / opt.evolve))
                )
                if random.uniform(0, 1) < crossover_rate:
                    crossover_point = random.randint(1, len(hyp_GA) - 1)
                    child = population[parent1_index][:crossover_point] + population[par
ent2_index][crossover_point:]
                else:
                    child = population[parent1_index]
                # Adaptive mutation rate
                mutation_rate = max(
                    mutation_rate_min, min(mutation_rate_max, mutation_rate_max - (gener
ation / opt.evolve))
                )
                for j in range(len(hyp_GA)):
                    if random.uniform(0, 1) < mutation_rate:
                        child[j] += random.uniform(-0.1, 0.1)
                        child[j] = min(max(child[j], gene_ranges[j][0]), gene_ranges[j][
1])
                next_generation.append(child)
            # Replace the old population with the new generation
            population = next_generation
        # Print the best solution found
        best_index = fitness_scores.index(max(fitness_scores))
        best_individual = population[best_index]
        print("Best solution found:", best_individual)
        # 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 generate_individual(input_ranges, individual_length):
    """
    Generate an individual with random hyperparameters within specified ranges.
```

```
    Args:
        input_ranges (list[tuple[float, float]]): List of tuples where each tuple contai
ns the lower and upper bounds
            for the corresponding gene (hyperparameter).
        individual_length (int): The number of genes (hyperparameters) in the individual
.

    Returns:
        list[float]: A list representing a generated individual with random gene values
within the specified ranges.

    Example:
        ```python
        input_ranges = [(0.01, 0.1), (0.1, 1.0), (0.9, 2.0)]
        individual_length = 3
        individual = generate_individual(input_ranges, individual_length)
        print(individual)  # Output: [0.035, 0.678, 1.456] (example output)
        ```

    Note:
        The individual returned will have a length equal to `individual_length`, with ea
ch gene value being a floating-point
        number within its specified range in `input_ranges`.
    """
    individual = []
    for i in range(individual_length):
        lower_bound, upper_bound = input_ranges[i]
        individual.append(random.uniform(lower_bound, upper_bound))
    return individual

def run(**kwargs):
    """
    Execute YOLOv5 training with specified options, allowing optional overrides through
keyword arguments.

    Args:
        weights (str, optional): Path to initial weights. Defaults to ROOT / 'yolov5s.pt
'.
        cfg (str, optional): Path to model YAML configuration. Defaults to an empty stri
ng.
        data (str, optional): Path to dataset YAML configuration. Defaults to ROOT / 'da
ta/coco128.yaml'.
        hyp (str, optional): Path to hyperparameters YAML configuration. Defaults to ROO
T / 'data/hyps/hyp.scratch-low.yaml'.
        epochs (int, optional): Total number of training epochs. Defaults to 100.
        batch_size (int, optional): Total batch size for all GPUs. Use -1 for automatic
batch size determination. Defaults to 16.
        imgsz (int, optional): Image size (pixels) for training and validation. Defaults
 to 640.
        rect (bool, optional): Use rectangular training. Defaults to False.
        resume (bool | str, optional): Resume most recent training with an optional path
. Defaults to False.
        nosave (bool, optional): Only save the final checkpoint. Defaults to False.
        noval (bool, optional): Only validate at the final epoch. Defaults to False.
        noautoanchor (bool, optional): Disable AutoAnchor. Defaults to False.
        noplots (bool, optional): Do not save plot files. Defaults to False.
        evolve (int, optional): Evolve hyperparameters for a specified number of generat
ions. Use 300 if provided without a
            value.
```

```
        evolve_population (str, optional): Directory for loading population during evolu
tion. Defaults to ROOT / 'data/ hyps'.
        resume_evolve (str, optional): Resume hyperparameter evolution from the last gen
eration. Defaults to None.
        bucket (str, optional): gsutil bucket for saving checkpoints. Defaults to an emp
ty string.
        cache (str, optional): Cache image data in 'ram' or 'disk'. Defaults to None.
        image_weights (bool, optional): Use weighted image selection for training. Defau
lts to False.
        device (str, optional): CUDA device identifier, e.g., '0', '0,1,2,3', or 'cpu'.
Defaults to an empty string.
        multi_scale (bool, optional): Use multi-scale training, varying image size by ±5
0%. Defaults to False.
        single_cls (bool, optional): Train with multi-class data as single-class. Defaul
ts to False.
        optimizer (str, optional): Optimizer type, choices are ['SGD', 'Adam', 'AdamW'].
 Defaults to 'SGD'.
        sync_bn (bool, optional): Use synchronized BatchNorm, only available in DDP mode
. Defaults to False.
        workers (int, optional): Maximum dataloader workers per rank in DDP mode. Defaul
ts to 8.
        project (str, optional): Directory for saving training runs. Defaults to ROOT /
'runs/train'.
        name (str, optional): Name for saving the training run. Defaults to 'exp'.
        exist_ok (bool, optional): Allow existing project/name without incrementing. Def
aults to False.
        quad (bool, optional): Use quad dataloader. Defaults to False.
        cos_lr (bool, optional): Use cosine learning rate scheduler. Defaults to False.
        label_smoothing (float, optional): Label smoothing epsilon value. Defaults to 0.
0.
        patience (int, optional): Patience for early stopping, measured in epochs withou
t improvement. Defaults to 100.
        freeze (list, optional): Layers to freeze, e.g., backbone=10, first 3 layers = [
0, 1, 2]. Defaults to [0].
        save_period (int, optional): Frequency in epochs to save checkpoints. Disabled i
f < 1. Defaults to -1.
        seed (int, optional): Global training random seed. Defaults to 0.
        local_rank (int, optional): Automatic DDP Multi-GPU argument. Do not modify. Def
aults to -1.

    Returns:
        None: The function initiates YOLOv5 training or hyperparameter evolution based o
n the provided options.

    Examples:
        ```python
        import train
        train.run(data='coco128.yaml', imgsz=320, weights='yolov5m.pt')
        ```


    Notes:
        - 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
    """
    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)
```

```
'''
Modified version of the loss.py script provided with the YOLOv5 model
Incorporated Weighted Cross-Entropy Loss

'''


import torch
import torch.nn as nn
import torch.nn.functional as F

from utils.general import bbox_iou
from utils.torch_utils import de_parallel

class ComputeLoss:
    def __init__(self, model, autobalance=False):
        device = next(model.parameters()).device
        h = model.hyp  # hyperparameters
        self.sort_obj_iou = h.get('sort_obj_iou')
        self.balance = [4.0, 1.0, 0.4, 0.1, 0.1]
        self.ssi = model.stride.argsort()[::-1]
        self.autobalance = autobalance
        self.na = model.model[-1].na
        self.nc = model.model[-1].nc
        self.device = device

        self.BCEcls = nn.BCEWithLogitsLoss(pos_weight=torch.tensor([16.0], device=device
))
        self.BCEobj = nn.BCEWithLogitsLoss(pos_weight=torch.tensor([1.0], device=device)
)

        # Focal Loss placeholder
        self.cp, self.cn = smooth_BCE(eps=h.get('label_smoothing', 0.0))
        g = h['fl_gamma']
        if g > 0:
            self.BCEcls, self.BCEobj = FocalLoss(self.BCEcls, g), FocalLoss(self.BCEobj,
 g)

        self.gr = 1.0
        self.hyp = h
        self.model = model

    def __call__(self, p, targets):
        device = self.device
        lcls = torch.zeros(1, device=device)
        lbox = torch.zeros(1, device=device)
        lobj = torch.zeros(1, device=device)
        bs = p[0].shape[0]  # batch size
        anchor = self.model.model[-1].anchors
        nl = self.model.model[-1].nl
        nt = len(targets)
        no = self.nc + 5
        balance = self.balance

        # Output targets
        tcls, tbox, indices, anchors = self.build_targets(p, targets)

        for i, pi in enumerate(p):
            b, a, gj, gi = indices[i]
            tobj = torch.zeros_like(pi[..., 0], device=device)  # target obj
```

```
            n = b.shape[0]
            if n:
                ps = pi[b, a, gj, gi]

                # Regression
                pxy = ps[:, :2].sigmoid() * 2. - 0.5
                pwh = (ps[:, 2:4].sigmoid() * 2) ** 2 * anchors[i]
                pbox = torch.cat((pxy, pwh), 1)
                iou = bbox_iou(pbox, tbox[i], xywh=True, CIoU=True)
                lbox += (1.0 - iou).mean()

                # Objectness
                iou = iou.detach().clamp(0).type(tobj.dtype)
                tobj[b, a, gj, gi] = iou  # iou ratio

                # Classification
                if self.nc > 1:
                    t = torch.zeros_like(ps[:, 5:], device=device)
                    t[range(n), tcls[i]] = 1.0
                    lcls += self.BCEcls(ps[:, 5:], t)
                else:
                    lcls += self.BCEcls(ps[:, 5], tcls[i].float())

            lobj += self.BCEobj(pi[..., 4], tobj) * balance[i]

        if self.autobalance:
            self.balance = [x / lobj.clone().detach().clamp(0.1) for x in balance]

        lbox *= self.hyp['box']
        lobj *= self.hyp['obj']
        lcls *= self.hyp['cls']
        bs = bs if isinstance(bs, int) else bs.shape[0]
        loss = lbox + lobj + lcls
        return loss * bs, torch.cat((lbox, lobj, lcls, loss)).detach()

    def build_targets(self, p, targets):
        raise NotImplementedError("Use original YOLOv5 build_targets() implementation")

class FocalLoss(nn.Module):
    def __init__(self, loss_fn, gamma=1.5, alpha=0.25):
        super().__init__()
        self.loss_fn = loss_fn
        self.gamma = gamma
        self.alpha = alpha

    def forward(self, input, target):
        loss = self.loss_fn(input, target)
        pt = torch.exp(-loss)
        return ((1 - pt) ** self.gamma * loss).mean()

def smooth_BCE(eps=0.1):
    return 1.0 - 0.5 * eps, 0.5 * eps
```