Code Report:
Multi-task Pose Estimation

Writer:

Amirshayan Nasirimajd

Supervisor:

Francesco Rossi

Ilaria Bloise

Date: 29/09/2022

Preprocess:

Main Function:

This function contains all the preprocessing steps.

The first part is to load the main information. First, we will load the configuration file containing all the configurations needed (read the Core section configuration) for doing the preprocessing and training. Using the configuration, the data path will be loaded.

In the next step, the JSON file containing the information of the data that is going to be preprocessed will be loaded. These data are the name of the image, quaternion, and position.

The next step is loading camera data which will be used to create the 2D key points, and then the 3D key points will be loaded.

After that, based on the path of the data in the dataset, the domain of data will be loaded; after that, the output files and paths of new preprocessed data will be loaded and based on the arguments given to the main function, the mask path input and output and their new size will be loaded.

At the end of this part, the CSV file contains the information on new labels that will be made in the output path of preprocessed data.

```
    update_config(cfg, args)

    datadir = os.path.join(cfg.DATASET.ROOT)

    # Read labels from JSON file
    jsonfile = args.jsonfile
    print(f'Reading JSON file from {jsonfile}...')
    with open(jsonfile, 'r') as f:
        labels = json.load(f) # list

    # Read camera
    camera = load_camera_intrinsics(cfg.DATASET.CAMERA)

    # Read Tango 3D keypoints
    keypts3d = load_tango_3d_keypoints(cfg.DATASET.KEYPOINTS) # (11, 3) [m
]

    # Where to save CSV?
    if cfg.DATASET.DATANAME == 'speed':
        domain, split = args.jsonfile.split('/')[-2:]
    elif cfg.DATASET.DATANAME == 'prisma25':
        domain, split = '', args.jsonfile
    elif 'shirt' in cfg.DATASET.DATANAME:
```

```
        traj, domain, split = args.jsonfile.split('/')
        domain = traj + '/' + domain
    else:
        raise NotImplementedError('Only accepting speedplus and prisma25')
    outdir = os.path.join(datadir, domain, 'labels')
    if not os.path.exists(outdir): os.makedirs(outdir)
    csvfile = os.path.join(outdir, split.replace('json', 'csv'))
    print(f'Label CSV file will be saved to {csvfile}')

    # Where to save resized image?
    imagedir = os.path.join(datadir, domain,
            f'images_{cfg.DATASET.INPUT_SIZE[0]}x{cfg.DATASET.INPUT_SIZE[1
]}_RGB')
    if not os.path.exists(imagedir): os.makedirs(imagedir)
    print(f'Resized images will be saved to {imagedir}')

    if args.load_masks:
        maskdir = os.path.join(datadir, domain,
            f'masks_{int(cfg.DATASET.INPUT_SIZE[0]/cfg.DATASET.OUTPUT_SIZE
[0])}x{int(cfg.DATASET.INPUT_SIZE[1]/cfg.DATASET.OUTPUT_SIZE[0])}')
        if not os.path.exists(maskdir): os.makedirs(maskdir)
        print(f'Resized masks will be saved to {maskdir}')

    # Open
    csv = open(csvfile, 'w')
```

The Second main part is starting to process the data using the data available in the JSON file. In the first step, based on the names available in the JSON file, the data will be read from their path, and based on the configuration output size, the images will be resized to their new size and will be written into the output path.

After this, the next step is based on the mask arguments. The masks will be loaded, then resized, and will be written into the output path.

Then is time to load labels. The quaternion and position will be loaded, and using them with 3D key points and Camera information, the 2D projection of the 3D points will be calculated over the image, and using them then, we calculate the bounding box base on the min and max in 2 coordinates of X and Y over the image.

Finally, the bounding box coordinates, quaternion, position, and 2D projected key points would be saved as the label of the image data in the CSV file at the output path.

```
    for idx in tqdm(range(len(labels))):

        # ---------- Read image & resize & save
```

```
        filename = labels[idx]['filename']
        image    = cv2.imread(os.path.join(datadir,cfg.DATASET.DATANAME, d
omain, 'images', filename), cv2.IMREAD_COLOR)
        image    = cv2.resize(image, cfg.DATASET.INPUT_SIZE)
        cv2.imwrite(os.path.join(imagedir, filename), image)

        # ---------- Read mask & resize & save
        if args.load_masks:
            mask = cv2.imread(os.path.join(datadir,cfg.DATASET.DATANAME ,
domain, 'masks', filename), cv2.IMREAD_GRAYSCALE)
            # print(os.path.join(datadir,cfg.DATASET.DATANAME , domain, 'm
asks', filename))
            mask = cv2.resize(mask, [int(s / cfg.DATASET.OUTPUT_SIZE[0]) f
or s in cfg.DATASET.INPUT_SIZE])
            cv2.imwrite(os.path.join(maskdir, filename), mask)


        # ---------- Read labels
        if args.load_labels:
            q_vbs2tango = np.array(labels[idx]['q_vbs2tango'], dtype=np.fl
oat32)
            r_Vo2To_vbs = np.array(labels[idx]['r_Vo2To_vbs_true'], dtype=
np.float32)

        # ---------- Project keypoints & origin
        if args.load_labels:
            # Attach origin
            keypts3d_origin = np.concatenate((np.zeros((3,1), dtype=np.flo
at32),
                                              keypts3d), axis=1) # [3, 12]

            keypts2d = project_keypoints(q_vbs2tango,
                                         r_Vo2To_vbs,
                                         camera['cameraMatrix'],
                                         camera['distCoeffs'],
                                         keypts3d_origin) # (2, 12)

            keypts2d[0] = keypts2d[0] / camera['Nu']
            keypts2d[1] = keypts2d[1] / camera['Nv']
            # Into vector (x0, y0, kx1, ky1, ..., kx11, ky11)
            keypts2d_vec = np.reshape(np.transpose(keypts2d), (24,))

        # ---------- Bounding box labels
        # If masks are available, get them from masks
        # If not, use keypoints instead
```

```
        if args.load_labels:
            xmin = np.min(keypts2d[0])
            ymin = np.min(keypts2d[1])
            xmax = np.max(keypts2d[0])
            ymax = np.max(keypts2d[1])
        # CSV row
        row = [filename]

        if args.load_labels:
            row = row + [xmin, ymin, xmax, ymax] \
                      + q_vbs2tango.tolist() \
                      + r_Vo2To_vbs.tolist() \
                      + keypts2d_vec.tolist()

        row = ', '.join([str(e) for e in row])

        # Write
        csv.write(row + '\n')

    csv.close()
```

Running the preprocess:

The main function of the preprocessing will be run on this part of the code. Here you are able to set the input arguments and set the list of The JSON files of different data splits.

```
if __name__=='__main__':
  files_name = ['validation.json','train.json','test.json']
  for file_name in files_name:
      args = easydict.EasyDict({
      'cfg' : 'experiments/offline_train_full_config_phi3_BN_speed.yaml',
      'jsonfile' : 'Datasets/speed/synthetic/'+file_name,
      'load_masks' : True,
      'load_labels' : True,
      'opts' : []
      })
      main(args)
print('done\n\n')
```

Training:

Main function:

In this function, first, we update the configuration base on the configuration file, then the output path log path will be defined, and the number of available Cuda cores.

Finally, the main_worker function, which is responsible for training, will be called.

```
    args = parse_args()
    update_config(cfg, args)

    cfg.defrost()
    cfg.DIST.RANK = args.rank
    cfg.freeze()

    _, output_dir, log_dir = \
        create_logger_directories(cfg, phase='train', write_cfg_to_file=Tr
ue)

    if args.gpu is not None:
        warnings.warn('You have chosen a specific GPU')

    ngpus_per_node = torch.cuda.device_count()

    main_worker(
          args.gpu,
          ngpus_per_node,
          args,
          output_dir,
          log_dir,
          overfit_limit
        )
```

main_worker function:

In the main_worker function in the first step, we will define the seed for training based on the available seed in the configuration file. Then the configuration will be updated.

Then the logger will be set up, and the main model will be built from the core.net package in the build.py script; then, based on the availability of the GPU, the device will be set to either Cuda or CPU, and then the model will be loaded to the device. In the next step, the model data will be saved and then used the get_dataloader function in the core.dataset package in the build.py script will return two data loaders for the train and validation data base on the CSV file path of the data split in the configuration

file. Finally, the optimizer and the scaler will be loaded from the configuration file for using mixed precision training.

```
    # Set all seeds & cudNN
    set_seeds_cudnn(cfg, seed=cfg.SEED)

    # GPU?
    args.gpu = gpu
    if args.gpu is not None:
        print(f'Use GPU: {args.gpu} for training')

    update_config(cfg, args)

    # setup logger
    logger = setup_logger(log_dir, args.rank, 'train', to_console=args.dis
tributed)

    # build network
    model = build_spnv2(cfg)

    # GPU device
    if args.gpu is not None:
        torch.cuda.set_device(args.gpu)
        device = torch.device('cuda', args.gpu)
    else:
        device = torch.device('cuda')


    model = model.to(device)

    # write model summary to file
    if args.rank == 0:
        write_model_info(model.module if args.distributed else model, log_
dir)

    # Dataloaders
    train_loader = get_dataloader(cfg,
                                  split='train',
                                  distributed=args.distributed,
                                  load_labels=True)
    val_loader = get_dataloader(cfg,
                                split='val',
                                distributed=args.distributed,
                                load_labels=True)

    # Optimizer & scaler for mixed-precision training
```

```
    optimizer = get_optimizer(cfg, model)
    scaler = get_scaler(cfg)  # None if cfg.FP16 = False, cfg.CUDA = False
```

In the next part of the main_worker function, if the resume function is available and there is a checkpoint file, it will be loaded as the last state of the model, and training will be continued from that state.

```
    # Load checkpoints
    checkpoint_file = osp.join(output_dir, f'checkpoint.pth.tar')
    if cfg.AUTO_RESUME and osp.exists(checkpoint_file):
        last_epoch, best_score = load_checkpoint(
            checkpoint_file,
            model,
            optimizer,
            scaler,
            device)
        begin_epoch = last_epoch
    else:
        begin_epoch = cfg.TRAIN.BEGIN_EPOCH
        last_epoch = -1
        best_score = 1e20
```

The next step will be loading some of the basic information needed to validate the results, which are the camera, and 3D key points.

```
    # For validation
    camera = load_camera_intrinsics(cfg.DATASET.CAMERA)
    keypts_true_3D = load_tango_3d_keypoints(cfg.DATASET.KEYPOINTS)
```

The next main step will be to do the training for several epochs in a loop, for which do the training on a specific number of epochs specified in the configuration file.

In the loop for the adjust_learning_rate will adjust the learning rate, which adjusts the learning rate based on a specific number of epochs specified on the configuration file by decreasing it based on multiplication to a value specified in the configuration file.

The next function in the loop is the do_train, which trains the model over one epoch and returns the mean losses of the train, which will be saved in the train_result list. The do_train function will be called from the trainer.py script in core.engine package.

After each training the model will be validated over the validation set using the do_valid function interface.py script from the core.engine package and the result of the validation process will be saved on the score variable, which is the mean pose loss over the validation set if the value is less than the best_score variable, which always contains the minimum value for the best validation.

In any case of having the minimum best score of validation we put the counter to zero. Counter is a variable that decreases at each epoch if there is no improvement in the validation of model and after a specific number of epochs which is in the overfit_limit variable if there is no improvement from t last best validation score, the training process will be stopped.

Moreover, after each validation, the model states will be saved as a check point, and if we have the best score in validation, the model will be saved as the best score.

```
    # ---------------------------------------
    # Main loop
    # ---------------------------------------
    score = best_score
    is_best = False
    is_final = False
    counter = 0
    for epoch in range(begin_epoch, cfg.TRAIN.END_EPOCH):
        print('')
        # Learning rate adjustment
        adjust_learning_rate(optimizer, epoch, cfg)

        # Single epoch training
        train_results.append(do_train(epoch,
                 cfg,
                 model,
                 train_loader,
                 optimizer,
                 log_dir=log_dir,
                 device=device,
                 scaler=scaler,
                 rank=args.rank))
        # Validate on validation set
        score = do_valid(epoch,
                         cfg,
                         model,
                         val_loader,
                         camera,
                         keypts_true_3D,
                         log_dir=None,
                         device=device)

        if score < best_score:
            best_score = score
            is_best = True
            counter = 0
        else:
```

```
            is_best = False
            counter += 1
        valid_results.append(score)

        # Save
        save_checkpoint({
            'epoch': epoch + 1,
            'backbone': cfg.MODEL.BACKBONE.NAME,
            'heads': cfg.MODEL.HEAD.NAMES,
            'state_dict': model.state_dict(),
            'best_state_dict': model.module.state_dict() if args.distribut
ed else model.state_dict(),
            'best_score': best_score,
            'optimizer': optimizer.state_dict(),
            'scaler': scaler.state_dict() if scaler is not None else None
        }, is_best, epoch + 1 == cfg.TRAIN.END_EPOCH or counter == overfit
_limit, output_dir)
        if counter == overfit_limit:
          break
```

Test:

Main Function:

The test script is to test the best-saved model during the training. As a result, the first step of the test is to set the test file in configuration equal to the best-achieved model base on the validation score during training.

```
test_model = osp.join(cfg.OUTPUT_DIR, cfg.TEST.MODEL_FILE)
if not osp.exists(test_model) or osp.isdir(test_model):
    test_model = 'outputs/efficientdet_d3/full_config/model_best.pth.tar'
cfg.defrost()
cfg.TEST.MODEL_FILE = test_model
```

In the next step of this script is first to build and then load the best model status into it.

```
model = build_spnv2(cfg)

# Load checkpoint
if cfg.TEST.MODEL_FILE:
    model.load_state_dict(torch.load(cfg.TEST.MODEL_FILE,
map_location='cpu'), strict=True)
    logger.info('   - Model loaded from {}'.format(cfg.TEST.MODEL_FILE))
model = model.to(device)
```

Then we load the test dataset and form the test CSV file, which its location is set in the configuration.

```
test_loader = get_dataloader(cfg, split='test', load_labels=True)
```

The final step is to do the test using again the do_valid function from the core.engine package in the interface.py script.

Core packages:

Introduction:

The core is the main package consisting of six sub-packages (config, dataset, engine, nets, solver, utils). Each of these directories will be discussed.

config:

This directory has two main scripts related to the configuration of the hyperparameters:

- **default.py:** This script contains all the default values of the configuration hyperparameters for the training. However, in this script, there is an update function that will update the values of configuration attributes based on the configuration file by getting the path of this file load it.
- **efficientdet.py:** This script contains all the hyperparameters needed for building the efficientdet backbone.

dataset:

The dataset directory contains all the related scripts for loading datasets and data augmentation. It contains two sub-directories (target_generators, transforms) beside two main scripts (build.py, SPEEDPLUSDataset.py).

- **target_generators:** this sub-folder contains heatmap_generator.py, which uses the key points of the object and model output result to create a heatmap for the heatmap head in our model. This goal is achieved by defining the HeatmapGenrator class and defining a call function for this class to calculate the heatmap.
- **Transforms:** this sub-folder has three main scripts (build.py, coarsedropout.py, randomsunflare.py). The main task in this directory is to define the augmentation methods.
    - **build.py:** this script contains the build_transforms function that defines the augmentation classes based on the configuration setting, and the result of this function is a list called transforms and contains all the augmentation methods that are going to be applied to data.
    - **coarsedropout.py:** Coarse Dropout of the rectangular regions in the image. Modified from official Albumentations implementation to restrict the dropout regions within the target bounding box.
    - **randomsunflare.py: s**imulates Sun Flare for the image from https://github.com/UjjwalSaxena/Automold--Road-Augmentation-Library Modified from official Albumentations implementation to restrict the flare location within the target's bounding box.
- **build.py:** this script contains two main functions:
    - **build_dataset:** this function will make dataset and data loader using the SPEEDPLUSDataset class in SPEEDPLUSDataset.py in the same directory. The input of this function is configuration, data split type, and the necessity of loading labels.
    - **get_dataloader:** this function will return the data loader and use the build_dataset to make the dataset and return the data loader as the output.

- **SPEEDPLUSDataset.py:** this script contains SPEEDPLUSDataset class that loads the dataset, and it will load the images, their labels mask, and all other related information to the data.

engine:

This directory contains three main scripts (adapter.py, inference.py, trainer.py), and they are responsible for train and validating the model.

- **inference.py:** this script is responsible for doing the test and validation of the model. The main function of this script is the do_valid function.

```
metric_names = []
if 'heatmap' in cfg.TEST.HEAD:
    metric_names += ['heat_eR', 'heat_eT', 'heat_pose']
if 'efficientpose' in cfg.TEST.HEAD:
    metric_names += ['effi_iou', 'effi_eR', 'effi_eT', 'effi_pose']
if 'segmentation' in cfg.TEST.HEAD:
    metric_names += ['segm_iou']
```

In this function, first, based on the configuration, all the metrics related to testing or validation will be set.

```
for name in metric_names:
    metrics[name] = AverageMeter(name, _get_metric_unit(name), ':6.2f')
```

The next step is to define an AverageMeter class for each metric from the core.utils package at utils.py script.

```
for idx, (images, targets) in enumerate(data_loader):
```

The next step is loading data from the dataset to make the prediction and calculate the related loss for each head. The targets contain the ground truth label for each batch of images.

```
with torch.no_grad():
    # Forward pass
    outputs = model(images,
                    is_train=False,
                    gpu=device)
```

The other main step is to feed the model with the image to receive its prediction as an outputs variable.

```
for i, name in enumerate(cfg.TEST.HEAD):
    iou, err_q, err_t, speed = None, None, None, None
```

After that, we start to calculate the loss related to each head of the model.

```
keypts_pr, q_pr, t_pr = solve_pose_from_heatmaps(
    outputs[i].squeeze(0).cpu(),
    cfg.DATASET.IMAGE_SIZE,
    cfg.TEST.HEATMAP_THRESHOLD,
    camera, keypts_true_3D
)
```

One of the most important functions to calculate the loss is related to the heatmap, which is the solve_pose_from_heatmaps function that tries to solve the PnP problem using key points and heatmap head outputs to find the position of the object.

- **trainer.py:** this script does the training process and defines the do_train function to do this task. For this purpose, first, we define the head as we did in the do_valid function.
  The next step is again to load the data from the data loader.
  And then calculate the loss and backpropagate the loss to the model to train it.

```
# Zero gradient
optimizer.zero_grad(set_to_none=True)

# Enable mixed-precision learning is scaler is provided
with autocast(enabled=scaler is not None):
    loss, loss_items = model(images,
                             is_train=True,
                             gpu=device,
                             **targets)

# Compute & update gradient
if scaler is not None:
    # Use mixed-precision
    scaler.scale(loss).backward()

    # Unscale before clipping
    scaler.unscale_(optimizer)
    clip_grad_norm_(model.parameters(), 1.0)
    scaler.step(optimizer)

    # Update the scale for next iteration
    scaler.update()
else:
    loss.backward()
    clip_grad_norm_(model.parameters(), 1.0)
    optimizer.step()

# Record losses
for k, v in loss_items.items():
  if k != 'ent':
        loss_meters[k].update(float(v), images.shape[0])
```

- **adapter.py**: The adapter is used in online domain refinement to do the domain refinement for our model.

nets:

This directory is responsible for making the SPNV2 model. It contains four sub-folder (backbones, heads, layers, and loss) and one main script, which is build.py.

- **backbones:** this sub-folder contains all the implementations of the backbones, which are three scripts:
  - **bifpn.py:** this script has the implementation of Bi-directional Feature Pyramid Network (BiFPN) adopted from the unofficial pytorch implementation at https://github.com/zylo117/Yet-Another-EfficientDet-Pytorch.
  - **efficientnet.py:** this is the implementation of the efficientnet, and copied and modified from the official PyTorch implementation (torchvision v0.11.0) to allow building with GN

for phi > 4 :
https://pytorch.org/vision/0.11/_modules/torchvision/models/efficientnet.html#efficientnet_b2

  - **efficientdet.py:** this is the implementation of the efficientdet model based on:
    - EfficientPose (official, TensorFlow): https://github.com/ybkscht/EfficientPose
    - EfficientDet (official, TensorFlow): https://github.com/google/automl
    - EfficientDet (unofficial, PyTorch): https://github.com/zylo117/Yet-Another-EfficientDet-Pytorch

- **heads:** this sub-folder contains all the implementations of the heads for the SPNv2 model. Which contains (anchors.py, efficientpose.py, heatmap.py, and segmentation.py).
- **layers:** This sub-folder contains the Conv layer needed for the efficientdet. And there is a file named conv.py that implemented the depthwise Separable Convolution layer for EfficientDet, implementation from https://github.com/zylo117/Yet-Another-EfficientDet-Pytorch/blob/master/efficientdet/model.py
- **loss:** this sub-folder contains loss functions scripts in different files such as ciou_loss.py for Intersection of Union loss, focal_loss.py for focal loss for object presence/absence classification, heatmap_loss.py for heatmap loss, speed_loss.py for pose loss, and transformation_loss.py for transformation loss.
- **build.py:** The build function is the main script that builds efficientdet backbone to get all relevant scaled parameters and adds related heads.

solver:

The solver contains only one file build, and it is responsible for several functions used during the training.

- **Solver:** the solver functions are:
  - **get_optimizer:** return the optimizer base on the configuration.
  - **get_scaler:** return scaler for mixed precision.
  - **adjust_learning_rate:** decay learning rate based on a schedule.

utils:

This directory contains five python files (checkpoints.py, metrics.py, postprocess.py, utils.py, and visualize.py) that have tools and functions that are used in every stage, such as preprocessing, training, or testing.

- **checkpoints.py:** this file has two functions, save_checkpoints and load_checkpoint. The save_checkpoints save the model's last state that can be loaded by using load_checkpoint.
- **metrics.py:** this file contains several functions that calculate the scores and errors for validation, such as IoU, IoU for segmentation, or pose score.
- **postprocess.py:** this file consists of several main functions used in different steps:
  - **solve_pose_from_heatmaps:** this function is used in validation to solve PnP problem based on the heatmap head predicted key points, and it uses the pnp function for this goal.

```
# PnP
q, t =
pnp(np.transpose(keypts_true_3D)[visibleIdx],keypts[visibleIdx].numpy(),
                camera['cameraMatrix'], camera['distCoeffs'])
```

  - **pnp:** this function solves the Perspective n points problem to predict the position of the objects based on the predicted key points of the object. This function is used by the solve_pose_from_heatmaps function.
  - **quat2dcm:** This function is used in to compute the direction cosine matrix from the quaternion.
  - **project_keypoints:** This function is used in preprocessing script to Project 3D key points to 2D and provide us with dataset labels.
- **utils.py:** This script contains some tools for calculating loading or writing and defining logs, some of the important functions and classes in this file are:
  - **AverageMeter Class:** it computes and stores the average and current value Modified from https://github.com/pytorch/examples/blob/master/imagenet/main.py
  - **ProgressMeter Class:** Prints training progress Modified from https://github.com/pytorch/examples/blob/master/imagenet/main.py
  - **load_tango_3d_keypoints**: This function loads the mat file containing all the 3D key points of the tango spacecraft.
  - **load_camera_intrinsics:** This function loads camera information from the camera.json file.
- **Visualize.py:** This file contains functions needed to visualize the image data the prediction results such as bounding box and etc.